VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE APLIKACÍ NA SUPERPOČÍTAČI POMOCÍ JAZYKA PYTHON

ACCELERATION OF APPLICATIONS ON A SUPERCOMPUTER USING PYTHON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MAREK ČELKA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARTA ČUDOVÁ**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Čelka Marek**
Obor: Informační technologie
Téma: **Akcelerace aplikací na superpočítači pomocí jazyka Python**
Acceleration of Applications on a Supercomputer Using Python
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s programovacím jazykem Python a jeho knihovnami numpy, mpi4py a matplotlib.
2. Seznamte se s paradigmaty paralelního zpracování a se standardem MPI.
3. Navrhněte a implementujte sadu mikrotestů s cílem vyhodnotit výkonnost MPI funkcí v jazyce Python.
4. Zvolte vhodný problém, který budete paralelizovat, vytvořte jeho paralelní řešení v jazyce Python a ověřte jeho funkčnost na superpočítači.
5. Ověřte výkonnost navržených řešení pomocí běžných metrik.
6. Zhodnoťte a diskutujte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Čudová Marta, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

V dnešnej dobe sú už všetky počítače, ktoré používame schopné paralelného spracovávania, ktoré nám výrazne šetrí čas pri riešení náročných problémov, ako sú napríklad rôzne vedecké výpočty, simulácie alebo predpovede. Témou tejto práce je akcelerácia výpočtovo náročných aplikácií práve pomocou paralelnému spracovania zadaného problému na superpočítači. Pre jednoduchšie pochopenie danej problematiky ľuďmi, ktorých sa priamo týka (napr. vedci, výzkumníci), bol zvolený programovací jazyk Python, ktorý je jednoduchý na pochopenie a mocný zároveň. Prvá časť práce sa venuje zoznámeniu s technikami paralelného spracovania pomocou mikrotestov, ktorých výsledky boli diskutované a využité ďalej v práci. Druhá časť práce sa zaoberá problematikou rekonštrukcie obrazu. Výsledky experimentovania s rôzne veľkými obrázkami sú porovnané v rámci sekvenčnej a paralelnej verzie pri rekonštrukcii obrazu a zápise do súboru. Spracované výsledky sú zhodnotené, diskutované a porovnané medzi sebou. Medzi použité metriky patria čas, zrýchlenie, priepustnosť a latencia.

Abstract

Nowadays, all computers we use are capable of parallel processing that saves time in compute-intensive tasks such as scientific computations, various simulations or predictions. The theme of this thesis is acceleration of compute-intensive tasks on supercomputer. This is achieved by the parallelization of the problem. For better understanding the issue by scientists from diverse scientific fields, the python programming language was chosen. Python is very powerful and easy to use as well. The first part of the thesis deals with the parallel processing techniques. The set of microtests was designed and implemented for this purpose. Results are then discussed and used in the further work. The second part of the thesis deals with the problem of parallel image reconstruction. For a comparison, the sequential version of the problem was also implemented. Both versions, sequential and parallel, were tested on a set of images of a different size. Experiments focus on acceleration, spent time, memory bandwidth and latency. These outcomes are also presented and discussed.

Kľúčové slová

MPI, mpi4py, python, numpy, superpočítač Anselm, formát pgm, detektor hrán, rekonštrukcia obrazu.

Keywords

MPI, mpi4py, python, numpy, supercomputer Anselm, format pgm, edge detector, image reconstruction.

Citácia

ČELKA, Marek. *Akcelerace aplikací na superpočítači pomocí jazyka Python*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marta Čudová

Akcelerace aplikací na superpočítači pomocí jazyka Python

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pani Ing. Marty Čudovej. Uviedol som všetky literálne pramene a publikácie, z ktorých som čerpal.

.....
Marek Čelka
17. mája 2017

Poďakovanie

Pri tejto príležitosti by som sa chcel veľmi pekne poďakovať vedúcej mojej práci pani Ing. Marte Čudovej za trpezlivosť a odborné rady, bez ktorých by vypracovanie tejto práce nebolo možné. Ďalej by som chcel poďakovať Bc. Dominikovi Šimekovi za poskytnuté odborné konzultácie. Túto prácu podporilo ministerstvo školstva, mládeže a športu veľké infraštruktúry pre výzkum, experimentálny vývoj a inovácie „IT4Innovations National Supercomputing Center – LM2015070“.

Obsah

1 Úvod **2**

2 Úvod do problematiky **3**
2.1 MPI . . . 3
2.2 Python . . . 5
2.3 Superpočítač Anselm . . . 5
2.4 Formát PGM . . . 8
2.5 Techniky paralelného spracovania . . . 9

3 Kritika súčasného stavu **13**

4 Návrh mikrotestov **15**

5 Implementácia mikrotestov **16**
5.1 Párová komunikácia . . . 16
5.2 Kolektívna komunikácia . . . 19
5.3 Násobenie matíc . . . 22

6 Návrh riešenia rekonštrukcie obrazu **27**
6.1 Detektor hrán . . . 28
6.2 Metóda rekonštrukcie . . . 28
6.3 Paralelná verzia rekonštrukcie . . . 29

7 Implementácia rekonštrukcie obrazu **30**
7.1 Sekvenčná verzia . . . 30
7.2 Paralelná verzia . . . 32
7.3 Zložitosť algoritmov rekonštrukcie obrazu . . . 34
7.4 Správa kódu . . . 35

8 Testovanie **36**
8.1 Meranie zrýchlenia rekonštrukcie . . . 36
8.2 Meranie zápisu do súboru . . . 38

9 Záver **39**

Literatúra **40**

A Obsah CD **42**

Kapitola 1

Úvod

Existujú situácie, kedy doba behu aplikácie alebo zložitejšieho výpočtu može trvať niekoľko dní až mesiacov (napr. 24-hodinová predpoveď počasia). V dnešnej dobe potrebujeme riešiť stále väčšie výpočtové problémy (napr. podrobnejšie a presnejšie modelovanie, simulácie a analýza). Pracujeme stále s väčším objemom dát a potrebujeme súbežný beh niekoľkých aplikácií. V týchto prípadoch potrebujeme zvýšením výpočtového výkonu ušetriť čo najviac, či už času alebo nákladov. Jednou z možností ušetrenia času, keď máme k dispozícií viacej jadier CPU je upraviť časti sekvenčného programu, ktoré môžu bežať súbežne do takzvaného paralelného programu. Ďalej je potreba zabezpečiť výkonný počítač, na ktorom vytvorený program pobeží. Dostatočný výkon poskytujú centrá, ktoré pracujú so superpočítačmi.

Veľké simulácie a iné programy náročné na výkon sú prevažne písané v programovacích jazykoch C/C++, či Fortran pre ich rýchlosť. Tieto jazyky sú však náročné na pochopenie pre neprogramátorov. Preto sa hľadá programovací jazyk, ktorý by bol rýchlejší a jednoduchší na pochopenie a zároveň vhodný na riešenie týchto úloh. Programovací jazyk Python je vďaka veľkému množstvu dostupných knižníc, dobrej čitateľnosti kódu a jednoduchosti implementácie jednou z možností. Keďže pri návrhu simulácií a ich implementácií spravidla spolupracujú vedci z rôznych odborov a programátori, jazyk ako Python by mohol vyplniť komunikačnú medzeru medzi nimi. Výzvou tejto práce je teda otestovať jazyk Python pre tento typ úloh s využitím konceptu MPI a porovnať ho s jazykom C.

V tejto práci sa budem venovať paralelnému spracovaniu niekoľkých problémov za účelom skrátenia času behu programu a efektívnejšieho využitia strojového času. Ako prvé budú prezentované základné techniky paralelného programovania pomocou jazyka Python na jednoduchých mikrotestoch. Na základe týchto mikrotestov bude identifikovaná z hľadiska výkonnosti najvýhodnejšia datová štruktúra v jazyku Python a porovnaná rýchlosť programu v jazykoch Python a C. Následne, s využitím takto získaných skúseností, sa v práci budem zaoberať komplexnejším problémom, tj. paralelná rekonštrukcia obrazu. Cieľom práce je potvrdiť alebo vyvrátiť hypotézu o vhodnosti jazyka Python v danej vednej oblasti. Experimentálne získané poznaky budú opäť prezentované a diskutované.

Kapitola 2

Úvod do problematiky

V nasledujúcej kapitole budú popísané nástroje a techniky paralelného spracovania použité pri tvorbe mikrotestov a rekonštrukcie obrazu. Cieľom mikrotestov bolo osvojiť si základy viacprocesového programovania pomocou štandartu MPI, porovnať náročnosť implementácie v jazyku C a Python, a efentivitu programov napísaných v týchto jazykoch. Experimentálne získané poznatky budú následne použité pri riešení paralelnej rekonštrukcie obrazu.

2.1 MPI

MPI (*message passing interface*) [12], [13] je štandart zabezpečujúci komunikáciu medzi procesy. Každý proces sa mapuje na jeden procesor. Procesy sú statické, tj. vytvoria sa v okamžiku načítania paralelného programu a existujú až do jeho ukončenia, ich počet je parametrom programu. Procesy sa združujú do skupín a sú identifikované na základe ich identifikačného čísla (*ranku*). Procesy operujú v rámci nezávislých adresových priestorov. Dáta niesú medzi procesmi zdielané, ale procesy komunikujú vysielaním/prijímaním správ cez prepojovaciu sieť. Podstatnou súčasťou MPI štandartu je komunikátor. Komunikátory sú skupiny procesov pri behu MPI programu. Komunikátor MPI_COMM_WORLD existuje vždy a obsahuje všetky procesy v rámci programu.

Komunikácia medzi procesmi môže byť buď párová alebo kolektívna. Jednotlivé typy komunikácií a ich funkcií budú otestované v rámci implementovaných mikrotestov v nasledujúcej kapitole 5.

Párová komunikácia

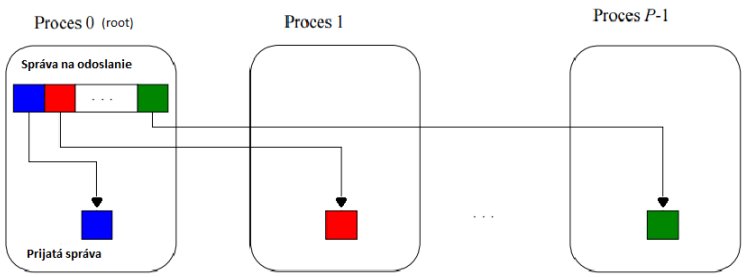
Párová komunikácia je komunikácia medzi dvoma procesmi. Odosielanie správ zaisťujú funkcie `send()` a prijímanie funkcie `recv()`. Komunikácia sa delí na [5]:

- Blokujúca (synchrónna) - Návrat z funkcie `send()` je až po úspešnom odoslaní správy z odosielacieho bufferu a z funkcie `recv()` po úspešnom prijatí správy do prijímacieho bufferu, kedy je možné bezpečne ďalej používať tieto buffre.
- Neblokujúca (asynchrónna) - Návrat z funkcii je okamžite. V tomto prípade je ale nutné pred nasledujúcim použitím bufferu použiť funkciu `MPI_Wait` (blokujúca) alebo `MPI_Test` (neblokujúca), pomocou ktorej sa zistí, či je daný buffer k dispozícií na čítanie, respektíve zápis.

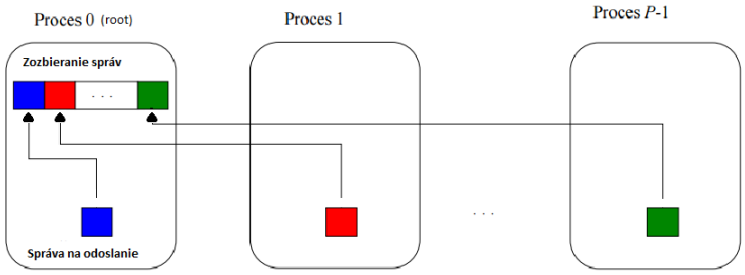
Kolektívna komunikácia

Kolektívna komunikácia je komunikácia medzi všetkými prosesmi v rámci komunikátoru. Funkcie kolektívnej komunikácie volajú všetky procesy komunikátoru, nasledujúce funkcie boli využité pri implementácií mikrotestov viď kapitola 5:

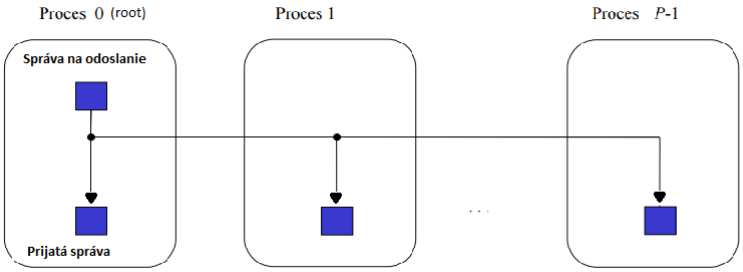
- Scatter - Jeden proces (root) je zdrojom dát, ktoré sú rozdelené rovnomerne medzi všetky procesy v komunikátore (obr. č. 2.1). Pri potrebe rozdeliť dáta nerovnomerne sa používá funkcia `Scatterv()`.
- Gather - Jeden proces (root) zozbiera dáta zo všetkých procesov (obr. č. 2.2).
- Broadcast - Jeden proces (root) je zdrojom dát, ktoré sú rozoslané všetkým procesom (obr. č. 2.3).
- Reduce - Funkcia `Reduce()` je klasický koncept z funkcionálneho programovania. Na základe nejakej funkcie (`sum(), multiply(), max(), ...`) prevedenej nad skupinou dát, je táto skupina zredukovaná do menšej skupiny dát [7].
- Barrier - Bariéra slúži na synchronizáciu procesov v rámci komunikátoru. Procesy sú zablokované na volanie tejto funkcie, pokiaľ ju nezavolajú všetky procesy komunikátora [6].



Obr. 2.1: Scatter: Hlavný proces (0) pomocou funkcie `Scatter()` rozdelí vstupné dáta rovnomerne medzi všetky procesy.



Obr. 2.2: Gather: Hlavný proces (0) pomocou funkcie `Gather()` zozbiera dáta zo všetkých procesov.



Obr. 2.3: Broadcast: Dáta sú pomocou funkcie `bcast()` zdielané medzi všetkými procesmi.

2.2 Python

Python je interpretovaný, objektovo orientovaný programovací jazyk. Vysoko-úrovňové datové štruktúry kombinované s dynamickým typovaním sú predurčené na rýchle vytváranie prototypov. Ďalšou vlastnosťou je jednoduchá možnosť rozšírenia. Nové zabudované moduly môžu byť jednoducho napísané v jazyku C či C++. Python je takisto multi-paradigmový jazyk, čo umožňuje používanie viacerých štýlov programovania. [11]

V nasledujúcich podkapitolách sú popísané hlavné moduly potrebné pre prácu s MPI v rámci programovacieho jazyka Python.

mpi4py

mpi4py [4] je knižnica, ktorá umožňuje využívať funkcie rozhrania Message Passing Interface (MPI) (podkapitola 2.1) v rámci jazyka Python. Je implementovaná podľa špecifikácie MPI-1/2/3 a poskytuje objektovo orientované rozhranie, ktoré sa podobá MPI-2 C++.

NumPy

NumPy [8] je základný balíček pre vedeckú prácu na počítači pomocou jazyka Python. Okrem iného obsahuje:

- výkonný N-dimenzionálny objekt poľa
- nástroj pre integráciu C/C++/Fortran kódu
- schopnosť pracovať s lineárnou algebrou, Fourierovou transformáciou, či náhodnými číslami

Okrem zjavného vedeckého uplatnenia, je možné NumPy využiť ako efektívny multi-dimenzionálny kontajner generických dát. Umožňuje definovať ľubovoľné datové typy, čo umožňuje rýchlu integráciu so širokou škálou databáz.

2.3 Superpočítač Anselm

Superpočítač [16] je všeobecné označenie počítača alebo počítačového systému s vysokým výkonom. Architektúra superpočítačov je vo väčšine prípadov postavená na jednoduchom princípe, kedy veľké množstvo spolupracujúcich jednoduchých elementov vytvára zložité

štruktúry. Superpočítače sa dnes využívajú takmer vo všetkých vedných oblastiach, či už predpovede počasia, modely kvantovej fyziky alebo kryptoanalýza. Superpočítač Anselm [3] je zoskupenie veľkého počtu výpočetných jednotiek do jednej formou počítačového clusteru. Pri programovaní sa používajú princípy distribuovaných a paralelných výpočtov.

Vysvetlenie základných pojmov využívaných pri práci na Anselme:

- Uzol - Reprezentuje jeden server obsahujúci dva procesory. V rámci superpočítača Anselm rozlišujeme dva typy uzlov:
  - Login uzol - uzol, na ktorý sa užívatelia prihlasujú pomocou `ssh`[1] (slúži len na prihlásenie, žiadne náročné operácie sa tu nespúšťajú).
  - Výpočetný uzol - uzol, na ktorom prebiehajú výpočty.
- Fronta - Slúži na správu poradia uživateľských úloh pre výpočet. Na Anselme je niekoľko typov front, ktoré majú rozdielne priority, potrebnú autorizáciu, maximálny počet zabraných uzlov, či maximálny čas behu programu:
  - Expresná - vyhradená pre beh a testovanie menších úloh. Vyššia priorita.
  - Produkčná - určená pre bežné produkčné využívanie. Stredná priorita.
  - Dlhá - určená pre produkčné, dlho bežiace úlohy. Stredná priorita.
  - Vyhradená - určená pre pripojenie na uzly s GPU kartami a Xeon Phi akcelerátormi. Veľmi vysoká priorita.
  - Voľná - pokiaľ sú vyčerpané vyhradené zdroje pre určitý projekt, tak je úloha zaradená do tejto fronty, ktorá využíva voľné zdroje superpočítača. Veľmi nízku prioritu.
- PBS (Portable Batch System) - Je počítačový software, ktorý má na starosti plánovanie a alokáciu potrebných zdrojov najčastejšie v prostredí UNIX clusterov.
- Moduly - Modul je súbor programov, zdieľaných knižníc a hlavičkových súborov potrebných k práci s požadovanými technológiami. Pred prácou na výpočetných uzloch je nutné najskôr načítať potrebné moduly. Napríklad pri načítaní modulu `openmpi` sa nastavia cesty ku kompilátoru, ku zdieľaným knižniciam a pripraví sa prostredie pre beh `mpi` programu. Pri tejto práci boli využité nasledujúce moduly s programami, knižnicami, prekladačmi, ...:
  - ScientificPython - modul na prácu s jazykom Python. Z hlavných knižníc, ktoré potrebujeme obsahuje `mpi4py` a `numpy`.
  - openmpi - modul na prácu s MPI v jazyku C.
- Job - Úloha, ktorá zahŕňa všetky potrebné procedúry (načítanie modulov, preklad, spustenie, ...) k spusteniu programu na Anselme. Pri podávaní žiadosti o zdroje Anselmu je potrebné špecifikovať minimálne:
  - Typ fronty
  - Počet výpočetných uzlov
  - Počet jadier na každý uzol

[1] SECURE SHELL - https://www.ssh.com/ssh/

- Maximálny čas behu Jobu (najneskôr po jeho vypršaní sa Job ukončí)
- ID projektu

Job (úlohu) podľa spôsobu spustenia delíme na:

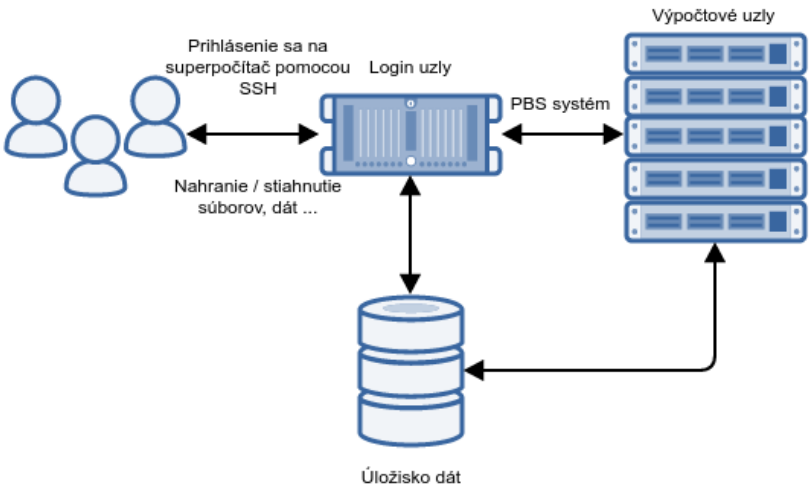
- Interaktívny - Ako prvé je potreba špecifikovať zdroje, čo odpovedá vyššie spomenutému výpisu. Za uvedenými zdrojmi je potrebné použiť prepínač `-I`, ktorý reprezentuje interaktívny job. Keď sú potrebné zdroje voľné, otvorí sa nový terminál v ktorom sa dá pracovať. Úlohy neni nutné zaraďovať do fronty, ale pracuje sa rovno s touto frontou.
- Job skript - Na základe Job skriptu, ktorý obsahuje informácie o tom, aké zdroje majú byť použité, čas behu programu a aj samotný program, je vytvorený Job identifikovaný pomocou ID. Job sa zaradí do zvolenej fronty a spustí sa, keď bude na rade a budú dostupné potrebné zdroje.

Postup práce na Anselme je zobrazený na obr. č. 2.4 a je nasledovný:

1. Prihlásenie sa na login uzol pomocou `ssh`.
2. Spustiť interaktívny job alebo vytvoriť job skript.
3. Pokiaľ sú požadované zdroje voľné, job sa vykoná. V prípade interaktívneho jobu sa uživateľovi vráti nový terminál s frontou a alokovanými zdrojmi, s ktorým môže pracovať po špecifikovanú dobu alebo pokým ho neukončí.

Technické parametre superpočítača Anselm [3]:

- 209 výpočetných uzlov:
  - 180x výpočetné uzly bez akcelerátora
  - 23x výpočetné uzly s GPU akcelerátormi NVIDIA Tesla Kepler K20
  - 4 výpočetné uzly s MIC akcelerátorom Intel Xeon Phi 5110P
  - 2 tučné uzly s 512 GB RAM a 100 GB SSD diskami
- Architektúra výpočetných uzlov: x86-64
- Uzle sú prepojené pomocou vysokorýchlostnej siete (InfiniBand)
- Topológia siete je tučný strom
- Operačný systém: Linux
- Počet jadier: 16 (2x8 jadier)
- RAM: 64 GB (4 GB na jadro)
- Lokálny disk: 500 GB



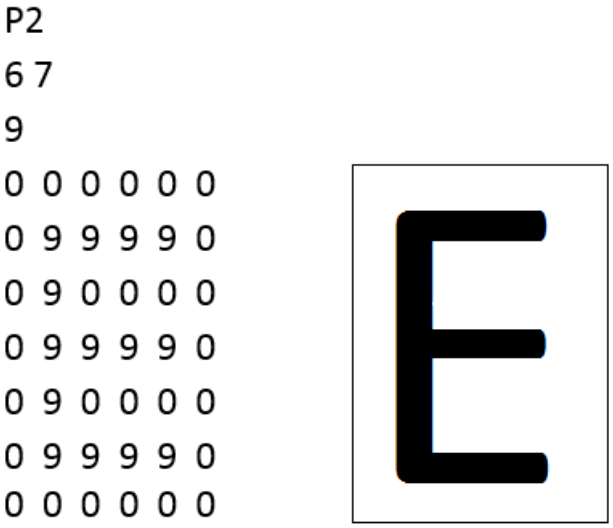
Obr. 2.4: Obecná schéma superpočítača Anselm: Uživatelia sa prihlasujú pomocou protokolu `ssh` na `login` uzle. Pomocou plánovača `PBS` sú užívateľmi zadané úlohy na spracovanie zaradené do fronty. Úložisko dát je dostupné z oboch typov uzlov.

2.4 Formát PGM

Formát PGM (portable graymap format) [9] je formát obrázkov, ktorý je navrhnutý na jednoduché zdieľanie medzi rôznymi platformami a jednoduché pochopenie. PGM obrázok predstavuje grafiku v odtieňoch sivej. Štruktúra PGM súboru (príklad PGM súboru na obr. č. 2.5):

1. magické číslo - udáva typ a kódovanie (ascii/binárne) formátu - pre PGM kódovanie ascii je magické číslo P2
2. biely znak
3. šírka - ASCII formát v desiatkovej sústave
4. biely znak
5. výška - ASCII formát v desiatkovej sústave
6. biely znak
7. ďalšia hodnota udáva maximálny počet hodnôt (stupňov sivej) medzi bielou (maximálna hodnota) a čiernou (hodnota 0)
8. biely znak (zvyčajne nový riadok)
9. nasleduje zápis hodnôt sivej reprezentujúcich obrázok o zadanej šírke a výške

riadky začínajúce znakom "#" sú považované za komentár.



Obr. 2.5: Príklad zápisu súboru podľa formátu PGM je možné vidieť na ľavej strane obrázka. Magické číslo P2 označuje použitý formát PGM, rozmery obrázka sú 6 x 7 pixelov a maximálna hodnota sivej je 9 (to znamená, že hodnota 9 reprezentuje čiernu a hodnota 0 bielu farbu, zvyšné hodnoty sú odtiene sivej). V pravej časti obrázka je možné vidieť zväčšenú, farebnú reprezentáciu bitmapy z ľavej časti.

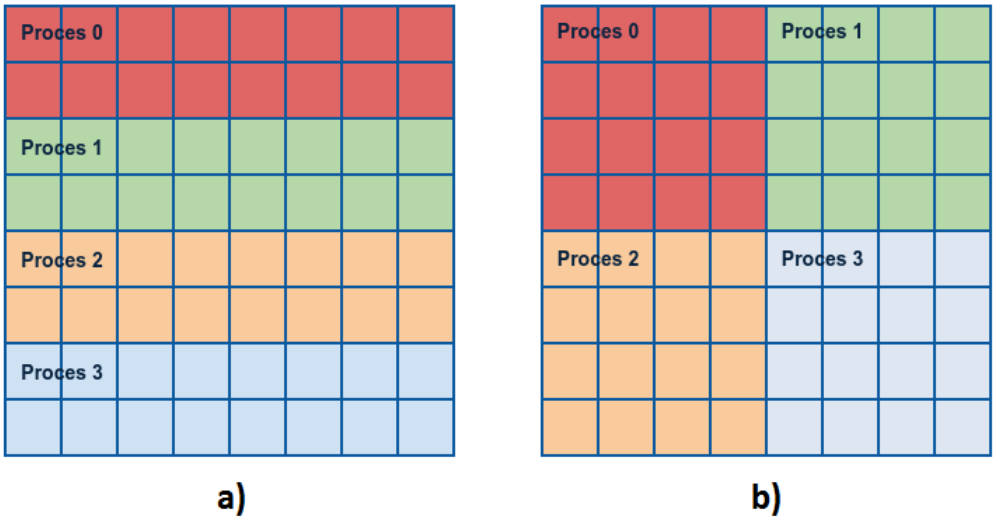
2.5 Techniky paralelného spracovania

Pre správny a efektívny návrh paralelného programu je dobré sa držať nasledujúcich bodov a podľa povahy riešeného problému zvoliť vhodný prístup pre paralelizáciu [14]:

1. Optimalizácia sekvenčnej verzie programu - Proces modifikácie programu za účelom jeho vyššej efektivity a zníženia zdrojových nárokov.
2. Dekompozícia problému na menšie časti - Analýza programu a identifikácia častí, ktoré by mohli bežať súčasne. Program sa delí na časť paralelizovateľnú a sekvenčnú. Dekompozícia sa delí na:
   - Funkčná - Minimalizácia závislosti (komunikácie) medzi úlohami. Čiastkových úloh by malo byť aspoň toľko, koľko jadier a mali by mať aspoň približne rovnakú veľkosť.
   - Datová - Cieľom je maximalizovať výpočetnú (aritmetickú) intenzitu – pomer výpočet/komunikácia. Je vhodná pri riešení problémov s veľkými dátami.
3. Identifikácia štruktúry programu:
   - Reťazové spracovanie - Softvérová reťazová linka pozostáva z niekoľkých procesov, kde výstup jedného je vstupom iného. Medzi susednými procesmi sú buffre vhodnej veľkosti. Predpoklady dobrého zrýchlenia:
     - Doba naplnenia/vyprázdnenia linky sa dá zanedbať
     - Pravidelný prísun veľkého množstva dát

  - Približne rovnaká doba spracovania v jednotlivých stupňoch
- Koordinácia udalosťami - Komponenty systému sú softvérové objekty, ktoré interagujú generovaním udalostí. Sú to správy s časovým razítkom. Tok dát nemá lineárnu štruktúru, nemusí byť jednosmerný, komunikácia je nepravidelná a nepredvídateľná. Udalosti definujú poradie spracovávania.
- Paralelizmus úloh - Problém ide rozdeliť na časti, ktoré môžu bežať súčasne. Rozdelenie úloh na jednotlivé procesy môže byť:
  - Statické mapovanie: Všetky úlohy sú známe a rozdelené procesom na začiatku tak, aby bola záťaž rovnomerne rozložená na všetky dostupné procesy.
  - Dynamické mapovanie: Používa sa v prípade, kedy úlohy vznikajú dynamicky alebo v prípade keď sa nemusia všetky úlohy dokončiť.
- Divide and conquer (Rozdeluj a panuj) - Rekurzívne delenie úlohy na menšie podúlohy až pokiaľ sa nedosiahne určitá prahová hodnota, kedy ďalšie delenie by už nebolo výhodné. Podúlohy sú rozdelené medzi procesy (veľkosť úlohy/počet procesov), ktoré ich vypočítajú. Potom sú výsledky podúloh spájané do výsledného riešenia.
- Rekurzívne dáta - Operácie s rekurzivnými dátovými štruktúrami (zoznam, strom, graf). Obvykle je potrebné pracovať so všetkými prvkami štruktúry paralelne, čo môže znamenať viacej práce, ale aj napriek tomu kratšiu dobu výpočtu. Používajú sa techniky napr. pointer jumping, recursive doubling.
- Geometrická dekompozícia - Hľadanie geometrického delenia spracovávaných dát. Rozdelenie dát na veľké množstvo nezávislých segmentov a riešenie hraníc medzi nimi. Vhodnými úlohami pre geometrickú dekompozíciu sú napríklad násobenie matíc alebo šírenie tepla. Príklad rozdelenia matice procesom (obr. č. 2.6):
  - 1D dekompozícia - rozdelenie matice procesom po riadkoch.
  - 2D dekompozícia - rozdelenie matice procesom po dlaždiciach.

  Po rozdelení dát je potreba komunikácie medzi procesmi na ich hraniciach, kde vznikajú nekonzistencie v dôsledku nedostatku informácií o okolitých segmentoch. Procesy si preto vymieňajú dáta prostredníctvom `Haló zón` znázornených na obrázku č. 2.7. Vymieňať si dáta v `Haló zónach` je možné každú iteráciu alebo raz za niekoľko iterácií. Rovnako je možné meniť veľkosť prekryvu, ktorý je v `Haló zónach` vymieňaný.
- Využitie podporných štruktúr kódu a dát (napr. SPMD, Master - worker, sdílená fronta, ...)



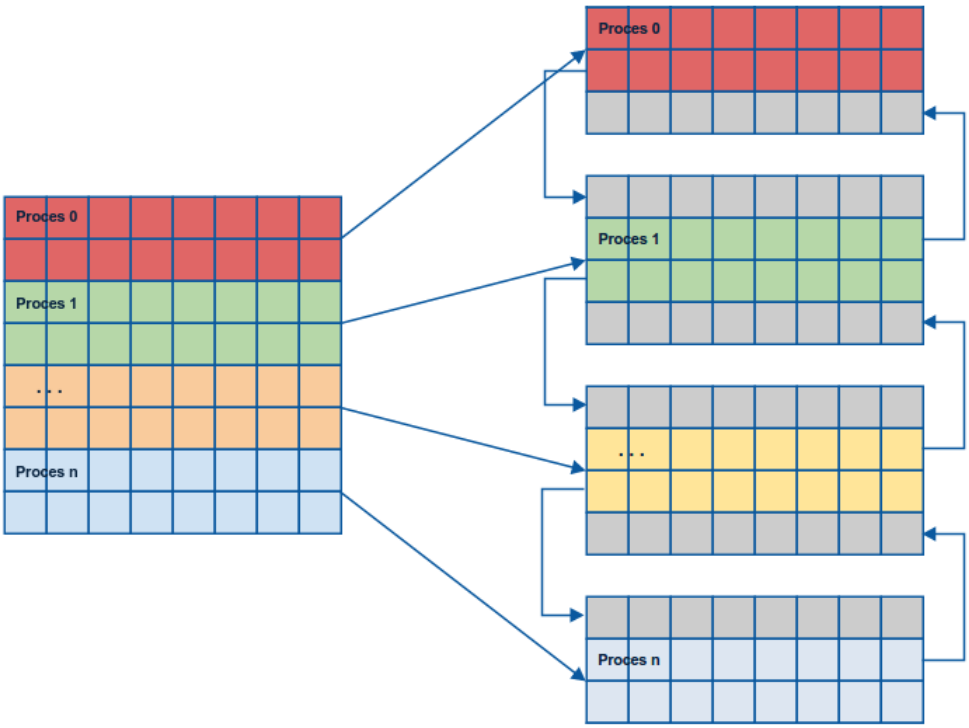
Obr. 2.6: Rozdelenie matice procesom: a) po riadkoch - 1D dekompozícia, b) po dlaždiciach - 2D dekompozícia.

4. Identifikácia kľúčových paralelných výpočtov - Identifikácia slúži na použitie známych vzorov (tried) výpočtov a komunikácie vyskytujúcich sa vo vedeckých, vstavaných i serverových aplikáciách (výpočet na mriežkach, časticové systémy, ...).

5. Implementácia kódu vo vhodnom programovacom prostredí (MPI [5], MKL[2], ...).

6. Ladenie výkonu.

[2] https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2017-release-notes



Obr. 2.7: Haló zóny: Na obrázku je možno vidieť prípad 1D dekompozície matice, kde je matica po riadkoch rozdelená medzi všetky dostupné procesy. Jednotlivé submatice sú rozšírené o veľkosť prekryvu (v tomto prípade je to jeden riadok). Toto rozšírenie sa nazýva `Haló zóna` a slúži na výmenu informácií so susednými procesmi. Tieto `Haló zóny` sú na obrázku znázornené sivou farbou a šípky naznačujú výmenu dát medzi procesmi.

Kapitola 3

Kritika súčasného stavu

V rámci 1. semestru bola navrhnutá a implementovaná sada mikrotestov v jazyku Python a C. Mikrotesty implementovali:

- Párovú komunikáciu
- Kolektívnu komunikáciu
- Násobenie matíc

Cieľom mikrotestov bolo získať základné znalosti písania paralelných programov, porovnať jazyky Python a C z hľadiska rýchlosti a získať vhodné dátové štruktúry pre jazyk Python. Z výsledkov mikrotestov vyplynulo, že jazyk Python je pomalší od jazyka C a vhodnou dátovou štruktúrov na prácu s dátami je štruktúra `numpy` polí. Detailnejšie informácie sa nachádzajú v kapitolách 4 a 5.

V druhom semestri bolo cieľom navrhnúť a implementovať komplexnejšiu úlohu, využiť poznatky z mikrotestov a overiť ich platnosť na väčšej úlohe. Ako vhodná úloha bola zvolená paralelná rekonštrukcia obrazu, ktorá pozostáva z nasledovných bodov:

- Vytvorenie návrhu rekonštrukcie obrazu.
- Vytvorenie vstupných dát: pre vstup rekonštrukcie je potrebné získať obrázky s detekovanými hranami. Takéto obrázky však nie sú bežne k dispozícií, preto prvým krokom implementácie bude vytvorenie vlastného detektora hrán.
- Obrázok detekovaných hrán vo formáte pgm bude vstupom pre rekonštrukciu.
- Pri rekonštrukcii obrazu bude použitá metóda inverzná k detekcii hrán - iteračná metóda počítania novej hodnoty zo 4-okolia.
- Implementácia sekvenčnej a paralelnej verzie v jazyku Python.
- Výstupom programu bude zrekonštruovaný obrázok vo formáte pgm.
- Testovanie bude prebiehať na superpočítači Anselm. Merania budú v rámci 1-n uzlov.
- Vyhodnotenie a diskusia výsledkov.
- Vytvorenie dokumentácie.

Pri venovaní sa práci v budúcnosti je možné prácu rozšíriť držaním sa nasledujúcich bodov:

- Implementácia rekonštrukcie pomocou 2D dekompozície obrazu.
- Implementácia farebného prevedenia rekonštrukcie.
- Experimentovanie s rôznymi okoliami pri počítaní novej hodnoty pixelu.
- Experimentovanie s Haló zónami - veľkosť prekryvu, výmena po X iteráciách, ...

Kapitola 4

Návrh mikrotestov

Jedným z cieľov návrhu a implementácie mikrotestov je osvojiť si základy písania a ladenia paralelných programov, ako aj experimentálne získať najvhodnejšiu dátovú štruktúru na prenos dát medzi procesmi v rámci programovacieho jazyka Python. Taktiež budú jednotlivé typy komunikácií medzi procesmi pomocou MPI porovnané medzi jazykmi Python a C. Porovnávané budú z hľadiska rýchlosti a jednoduchosti implementácie. Testovanie prebehne na superpočítači Anselm.

Prvá časť mikrotestov bude venovaná kolektívnej a párovej komunikácií medzi procesmi. V rámci párovej komunikácie bude implementovaná blokujúca a neblokujúca komunikácia v jazykoch Python a C. Test kolektívnej komunikácie bude zameraný na funkcie `Scatter()`, `Gather()` a `Reduce()`.

Druhá časť testov sa bude zaoberať implementáciou sekvenčnej a paralelnej verzíi násobenia dvoch matíc. V prvom kroku budú výsledky porovnané s výsledkami vstavanej funkcie knižnice `numpy` na vynásobenie matíc `dot()`. V ďalšom kroku budú porovnané sekvenčná verzia s paralelnou z hľadiska rýchlosti výpočtu. Všetky výsledky budú diskutované a zakreslené do grafov.

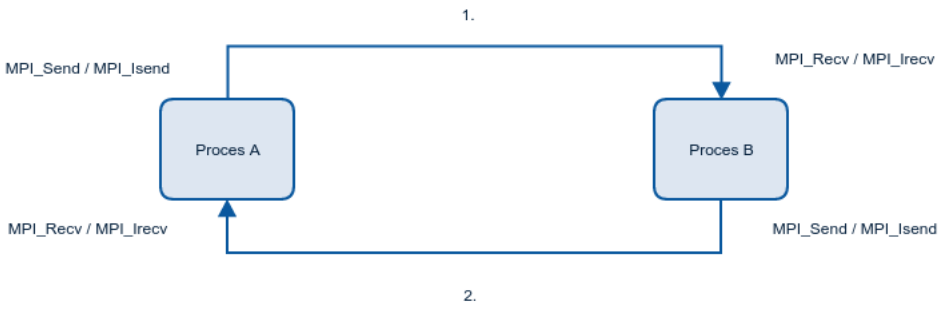
Kapitola 5

Implementácia mikrotestov

V rámci mikrotestov boli implementované programi na otestovanie párovej a kolektívnej komunikácie. Implementované boli v jazykoch Python a C. Názvy nasledujúcich podkapitôl zodpovedajú zameraním mikrotestov.

5.1 Párová komunikácia

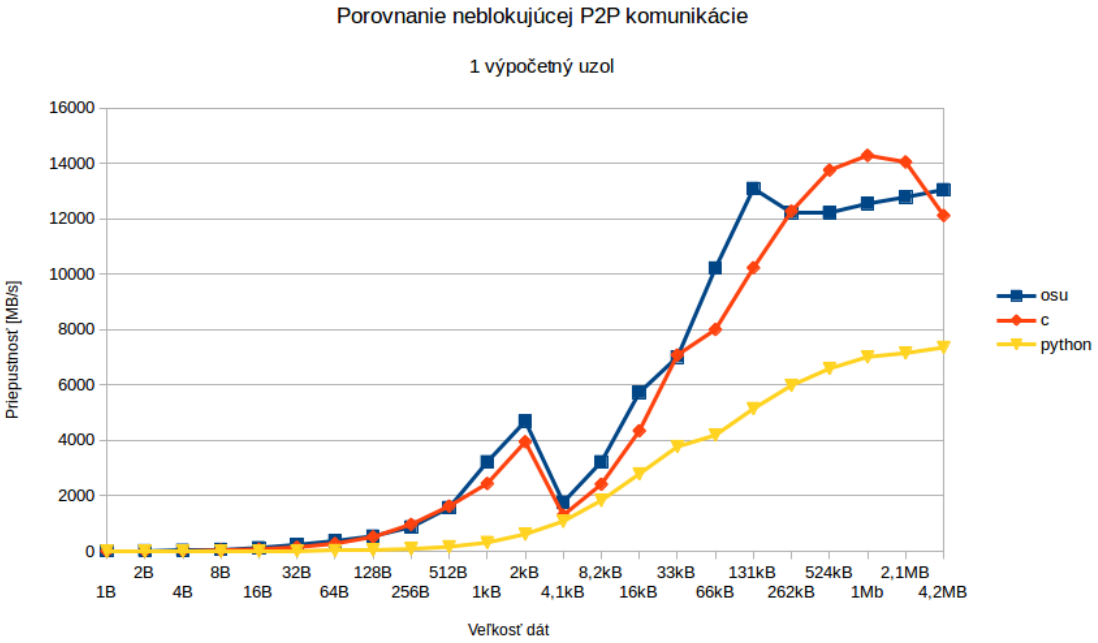
Program párovej komunikácie testuje blokujúcu a neblokujúcu komunikáciu medzi dvoma procesmi. Vstupom programu je typ komunikácie, veľkosť prenášaných dát a počet iterácií preposielania. Jedna iterácia (obr. 5.1) spočíva v poslaní správy z procesu A do procesu B a naspať z procesu B do procesu A. Tento proces je zmeraný pomocou funkcie `MPI_Wtime()` a vypísaný na štandartný výstup.



Obr. 5.1: Jedna iterácia párovej komunikácie. 1. Proces A pošle správu procesu B, 2. Proces B pošle prijatú správu naspäť procesu A.

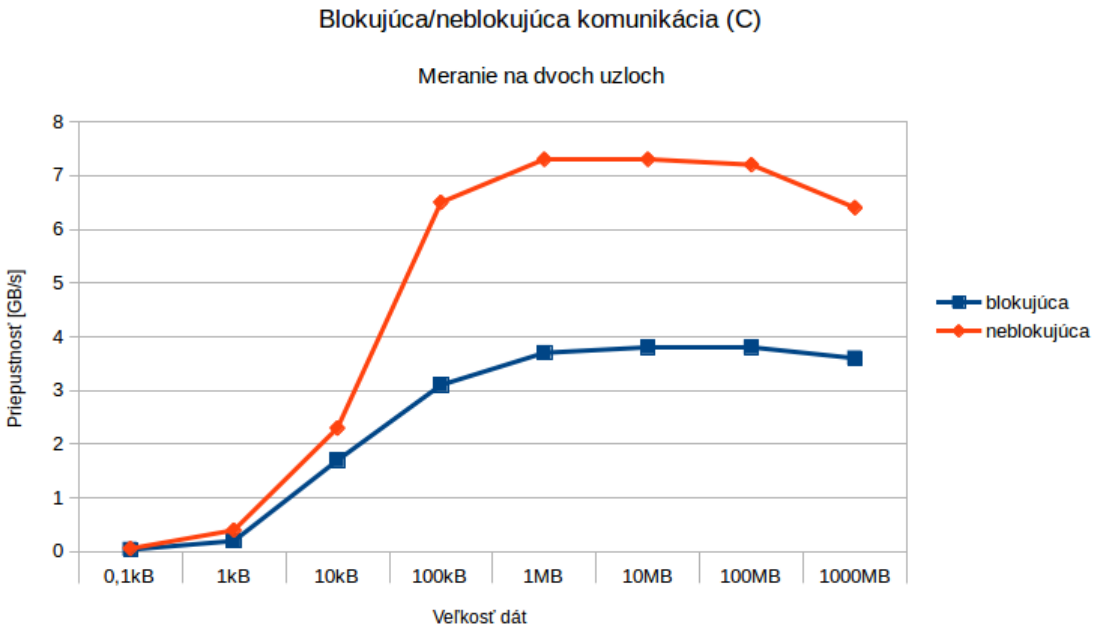
V tejto časti bola testovaná rýchlosť posielania dát pri blokujúcej a neblokujúcej komunikácií. Pri blokujúcej komunikácií posiela a prijíma správy v jednu chvílu iba jeden proces (program čaká na dokončenie funkcií `send()/recv()`), zatiaľ čo pri neblokujúcej komunikácií posielajú a prijímajú procesy správy súčasne (program je zablokovaný len funkciou `Wait()`, kedy je treba prijaté dáta odoslať naspäť). K tomu aby boli získané porovnateľné hodnoty sú správy posielané v niekoľkých iteráciách v závislosti na veľkosti prenášaných správ, aby bol čas pri meraniach približne rovnaký. Program bol implementovaný v jazyku

Python a v jazyku C. Výkonnosť tejto implementácie bola porovnaná s implementáciou OSU-Benchmark[1] v rámci neblokujúcej komunikácie (v rámci iných komunikácií bola implementácia mierne odlišná, tak porovnanie v týchto prípadoch by nedávalo zmysel). Z meraní (obr. č. 5.2) vyplýva, že priemerná priepustnosť implementácie OSU benchmarku je v porovnaní s mojou implementáciou približne 1,05 krát väčšia. V nasledujúcich grafoch na obr. č. 5.2, 5.3 a 5.4 sú teda porovnané jednotlivé jazyky z hľadiska rýchlosti pri rozdielnych veľkostiach dát, jednotlivé typy komunikácií a použité dátové štruktúry v jazyku Python. Testy boli vykonané v rámci jedného a dvoch výpočetných uzlov superpočítača Anselm.

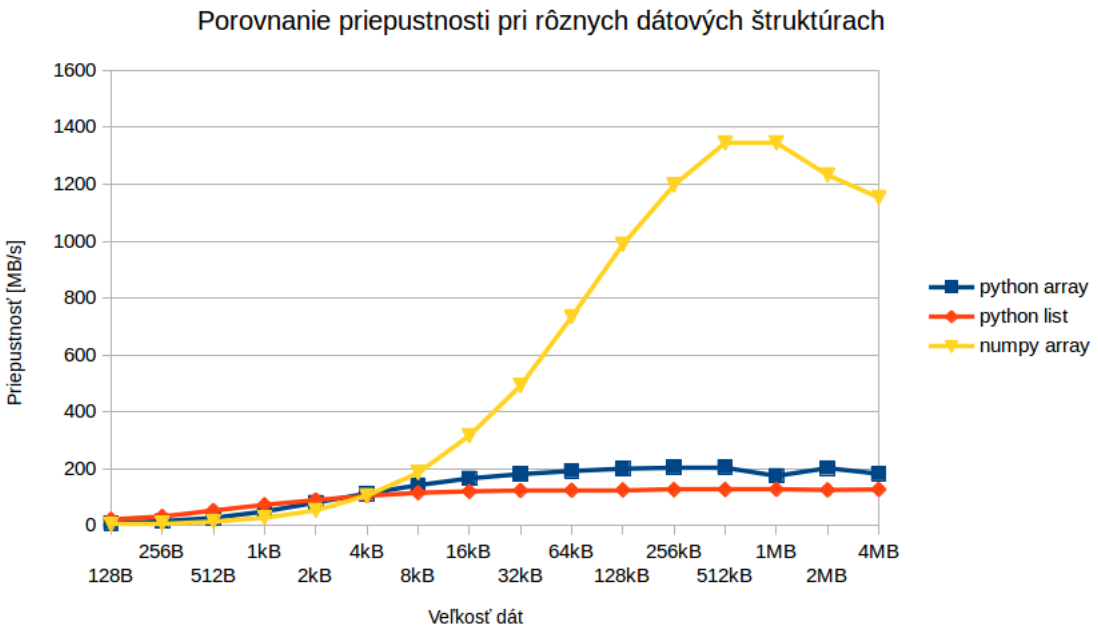


Obr. 5.2: Porovnanie neblokujúcej komunikácie medzi dvoma procesmi na jednom výpočetnom uzle superpočítača Anselm. Porovnávaná bola implementácia pomocou jazyka C a Python s OSU-Benchmarkom (`osu_bw`). Z grafu je možné usúdiť, že priepustnosť v jazyku Python stúpa asi 2 krát pomalšie, ako pri implementáciách v jazyku C.

[1] http://mvapich.cse.ohio-state.edu/benchmarks/



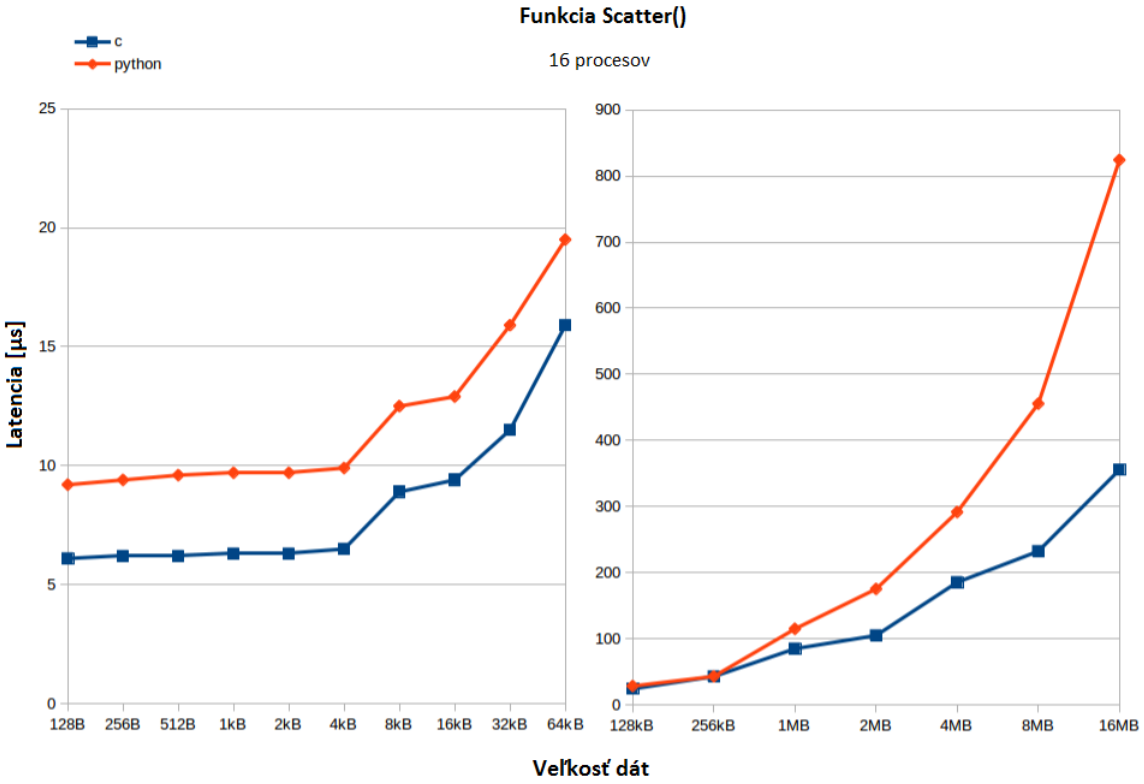
Obr. 5.3: Porovnanie blokujúcej a neblokujúcej komunikácie medzi dvoma procesmi na dvoch výpočetných uzloch. Na grafe je možné vidieť, že neblokujúca komunikácia je približne 2 krát rýchlejšia, ako blokujúca, pretože pri neblokujúcej sa posielajú dáta medzi procesmi súbežne (Full duplex) a pri blokujúcej posiela v jednej chvíly iba jeden proces (Half duplex). K maximálnej priepustnosti superpočítača Anselm (3.6 GB/s) sme sa priblížili pri veľkosti dát 10MB.



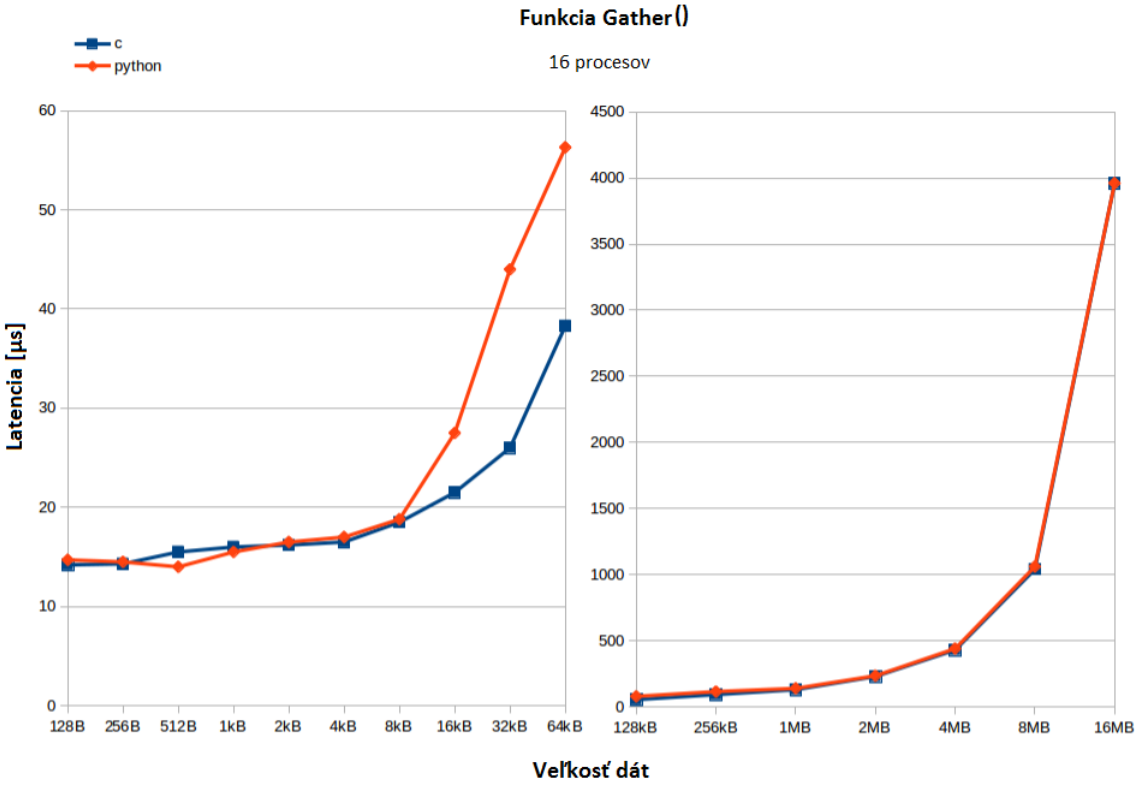
Obr. 5.4: Porovnanie dátových štruktúr v Pythone z hľadiska rýchlosti. Porovnávané boli dátové štruktúry Python pole (array), Python zoznam (list) a NumPy pole (array). Z grafu vyplýva, že najvhodnejšou (najrýchlejšou) dátovou štruktúrou je NumPy pole. K maximálnej dosiahnutej priepustnosti sa blížime pri veľkosti prenášaných dát 1MB.

5.2 Kolektívna komunikácia

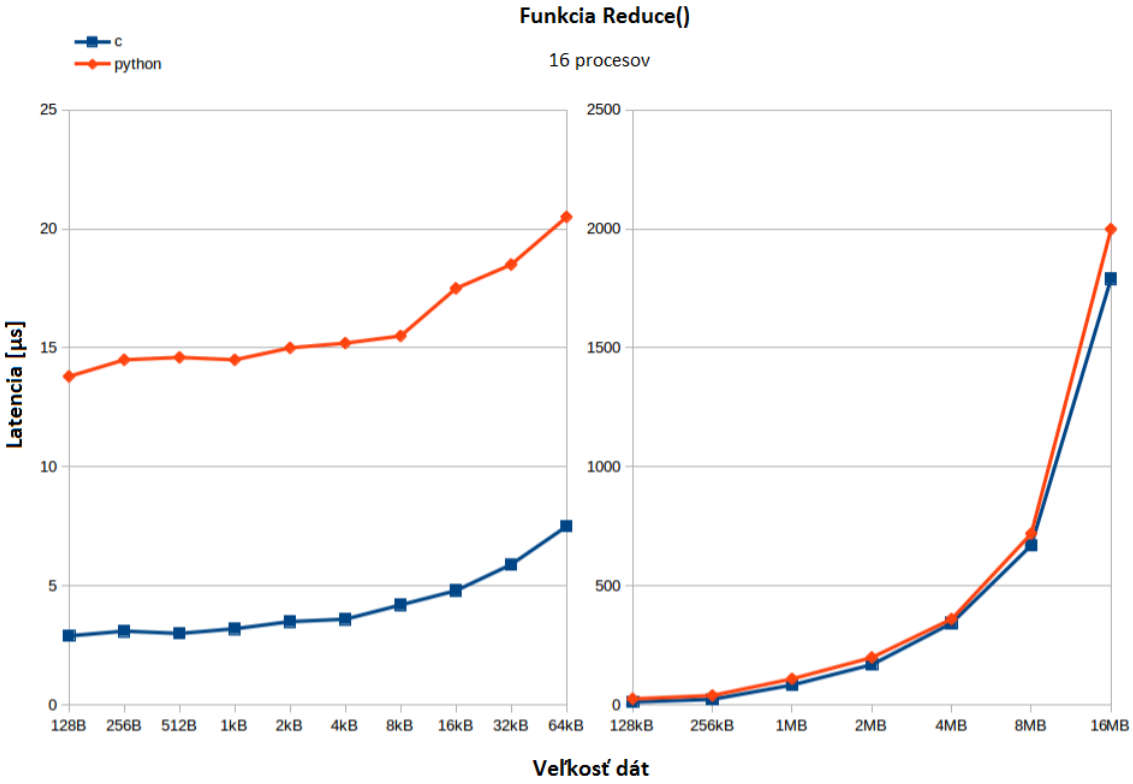
Vstupom programu na testovanie kolektívnej komunikácie je veľkosť dát a počet iterácií. Jedna iterácia pozostáva z rozdelenia poľa o zadanej veľkosti hlavným procesom pomocou funkcie `Scatter()` medzi všetky procesy, každý proces nastaví hodnoty prijatej časti poľa podľa svojho ranku a pomocou funkcie `Reduce()` vypočíta maximum, minimum a sumu tohoto poľa. Následne sú lokálne polia zo všetkých procesov zozbierané pomocou funkcie `Gather()` do výsledného poľa. Pre merateľné výsledky je použité väčšie množstvo iterácií, pretože jedna iterácia kolektívnej komunikácie trvá relatívne krátky čas. Nasledujúce grafy na obr. č. 5.5, 5.6 a 5.7 obsahujú porovnanie latencií jednotlivých častí programu (funkcie: `Scatter(), Reduce() a Gather()`) v jazyku Python a C pri rôznych veľkostiach dát (grafy boli pre lepšiu čitateľnosť rozdelené na 2 subgrafy).



Obr. 5.5: Porovanie funkcie `Scatter()` v jazykoch Python a C. Z grafu vyplýva, že pri menších dátach (ľavá časť grafu) je latencia funkcie `Scatter()` implementovaná v jazyku C približne konštantne 1,5 krát nižšia v porovnaní s implementáciou v jazyku Python. Pri zvyšovaní veľkosti dát od určitej veľkosti (pravá časť grafu) sa rýchlosť implementácie v jazyku C v porovnaní s jazykom Python postupne zväčšuje (v našom prípade na latenciu až 2,3 krát nižšiu).



Obr. 5.6: Porovnanie implementácie funkcie `Gather()` v jazykoch Python a C. Z grafu vyplýva, že pri menších dátach (ľavá časť grafu) je latencia implementácie v jazyku C priemerne 1,3 krát menšia a so zvyšujúcou sa veľkosťou dát (pravá časť grafu) sa rýchlosť jazykov Python a C vyrovnáva.



Obr. 5.7: Porovnanie implementácie funkcie `Reduce()` v jazykoch Python a C. Pri operáciách funkcie `Reduce()` je pri menších dátach (ľavá časť grafu) latencia implementácie v jazyku Python v priemere 3,5 krát väčšia, ako v jazyku C. Pri väčších dátach (pravá časť grafu) sa jazyky z hľadiska rýchlosti vyrovnávajú (latencia C implementácie je nižšia v priemere už len 1,15 krát).

5.3 Násobenie matíc

Posledný mikrotest sa zaoberal násobením matíc. Vstupom programu je veľkosť štvorcových matíc, ktoré chceme vynásobiť. Hlavný proces programu vygeneruje dve matice o zadanej veľkosti a podľa počtu dostupných procesov ich vynásobí, buď sekvenčne (1 proces) alebo paralelne (2 a viac procesov). Paralelný algoritmus bol implementovaný na základe veľkosti vstupných matíc:

- Ak to dovoľuje počet dostupných procesov a zadaná veľkosť matíc, matice A a B sú rozdelené pomocou 2D dekompozície rovnomerne na submatice (veľkosť submatice je $N/\sqrt{P}$, kde N je rozmer matice A a P je počet dostupných procesov) medzi procesy. Následne je použitý algoritmus č. 1 (*Násobenie na mriežke procesorov*) na vynásobenie matíc. V prvom kroku je potrebné previesť počiatočnú rotáciu matíc pred prvým vynásobením. Následne sa submatice navzájom vynásobia. Prebehne rotácia matíc (submatice matice A rotujú po riadkoch smerom doľava a submatice matice B rotujú po stĺpcoch smerom nahor) podľa počtu riadkov matice A. Jednotlivé medzivýsledky násobenia submatíc sa sčítavajú do výslednej matice C. Komunikácia medzi procesmi

je zabezpečená kombináciou neblokujúcej funkcie na posielanie správ `isend()` a blokujúcej funkcie na prijímanie správ `recv()`. Čiže po prijatí dát proces nečaká na odoslanie jeho dát, ale hneď pokračuje vo výpočte.

Algoritmus 1: Paralelné násobenie matíc [15].

Input : Matica A, Matica B
Output: Matica C

1 Rozdelenie matíc na štvorcové bloky podľa počtu dostupných procesov P;
2 Vytvorenie lokálnych matíc o veľkosti $P^{1/2}$ x $P^{1/2}$ prichystaných na blok matice A a B;
3 **repeat**
4 Jednotlivé bloky sú rozdelené procesom a lokálne sú vynásobené submatice A a B;
5 Výsledky sú uložené do lokálnej matice C;
6 Sub-matice A sú posunuté smerom do ľava a submatice B sú posunuté smerom nahor;
7 **until** *aktuálna iterácia* $< \sqrt{P}$;

- Ak nie je možné matice A a B rozdeliť na submatice, je použitá 1D dekompozícia, teda matica A je rozdelená po riadkoch pomocou funkcie `Scatter()` medzi dostupné procesy a matica B je broadcastovaná hlavným procesom medzi všetky procesy. Nakoniec sú všetky medzivýsledky zozbierané pomocou funkcie `Gather()` do výslednej matice C.

Validácia správnosti výsledku pri násobení matíc bola prevedená prostredníctvom porovnania výsledkov z vyššie naimplementovaných algoritmov s výsledkom funkcie `dot()` obsiahnutej v module `numpy` (popísaný v podkapitole 2.2). Táto funkcia príjme na vstupe dve matice, ktorých súčin vráti na jej výstupe.

Zložitosť algoritmov násobenie matíc

Sekvenčný algoritmus 2, pri ktorom počíta jeden proces skalárny súčin pre každý (celý) riadok matice A a každý (celý) stĺpec matice B má zložitosť:

$$t(N) = N \times N \times N = \mathrm{O}(N^3)$$

kde N je rozmer matice v jednej dimenzii (prepokladajme štvorcové matice).

Paralelný algoritmus č. 1 *Násobenie na mriežke procesorov* (2D dekompozícia) násobí matice pomocou P procesov. Počet procesov musí byť mocnina čísla dva a súčasne, rozmer matice N (v každej dimenzii) musí byť deliteľný $\sqrt{P}$. Matice A, B sa na začiatku algoritmu rozdelia na P procesov, každý proces teda dostane 1 submaticu o veľkosti $N/\sqrt{P} \times N/\sqrt{P}$ prvkov. Všetky procesy súčasne (paralelne) násobia svoje submatice pomocou sekvenčného algoritmu 2. V jednom kroku algoritmu sa teda paralelne vynásobí P submatíc (sekvenčným algoritmom). Po výmene dát medzi procesmi sa tento krok opakuje $\sqrt{P}$ krát. Zložitosť tohoto algoritmu je teda:

$$t(N) = \sqrt{P} \times (N/\sqrt{P})^3 = N^3/P$$

Ako príklad uvažujme počet procesov $P = N$ (N je mocnina čísla 2). V tomto prípade:

$$t(N) = N^3/P = N^3/N = \mathrm{O}(N^2)$$

Samozrejme, $P = N$ je extrémny prípad, pre veľké matice by bol potrebný veľmi veľký počet procesov, čo zahŕňa aj veľkú komunikačnú réžiu. Preto je nutné s počtom procesov experimentovať a nájsť hodnotu, kedy bude výpočet čo najefektívnejší.

Paralelný algoritmus (1D dekompozícia), kde matica B je rozhlásená medzi všetky procesy a matica A je rozdelená medzi P procesov po riadkoch funguje nasledovne:

- Každý proces obdrží celú maticu B a N/P riadkov matice A.
- Pre každý riadok matice A (N/P riadkov) sa počíta skalárny súčin s každým stĺpcom matice B, pričom hneď dostávame výsledný prvok matice C.

Zložitosť algoritmu je teda:

$$t(N) = N^2 \times N/P = N^3/P$$

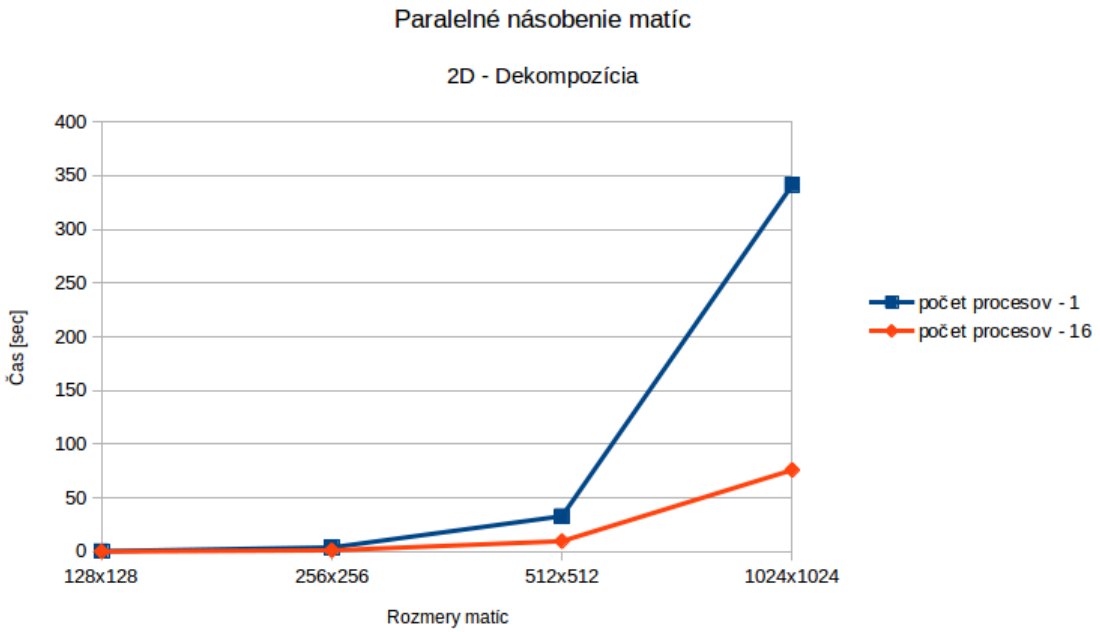
Ako príklad uvažujme počet procesov $P = N$. V tomto prípade je situácia podobná ako u predchádzajúceho algoritmu:

$$t(N) = N^3/P = N^3/N = \mathrm{O}(N^2)$$

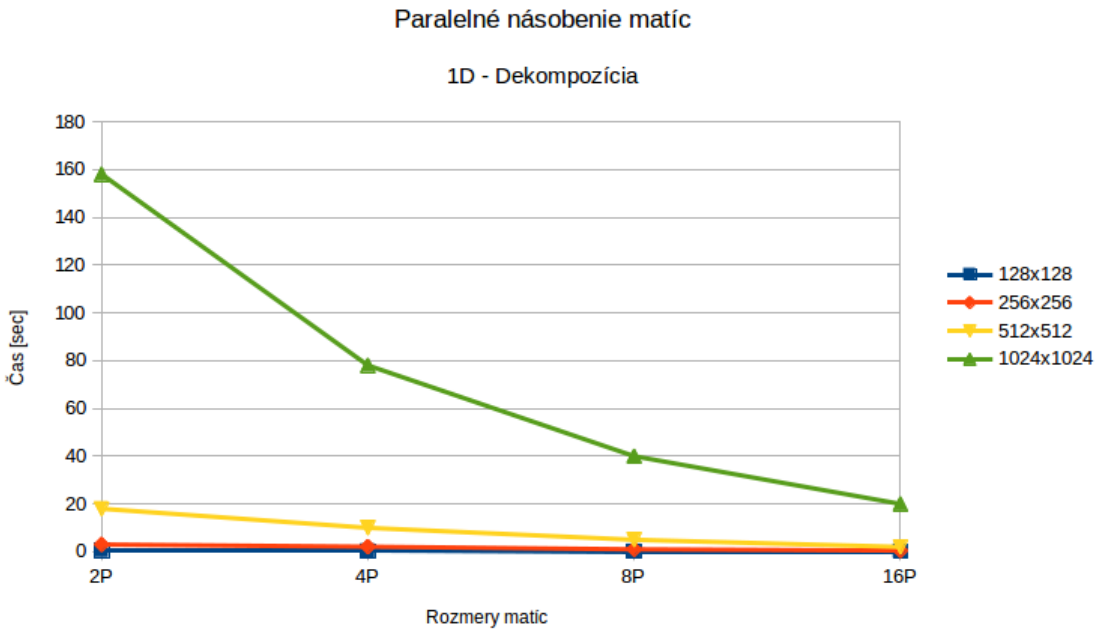
Algoritmus 2: Sekvenčné násobenie matíc.

```
   Input  : Matica A (m*k), Matica B(k*n)
   Output: Matica C(m*n)
1  for i in (0..m) do
2      for j in (0..k) do
3          dot = 0.0;
4          for x in (0..n) do
5              dot += A[i][x] * B[x][j];
6              C[i][j] = dot;
7          end
8      end
9  end
```

Na nasledujúcich grafoch (obr. č. 5.8 a 5.9) je porovnaný čas potrebný na vynásobenie dvoch matíc zadanej veľkosti pomocou jednotlivých algoritmov a rozdielným počtom dostupných procesorov.



Obr. 5.8: Metóda rozdelenia matíc A a B na submatice. Meranie prebiehalo v rámci jedného uzla superpočítača Anselm na štvorcových maticiach s rozmermi od 128x128 po 1024x1024. Pri použití šestnástich procesov prichádza k výraznému šetreniu času potrebného na výpočet (až 4,6 krát menej) v porovnaní s jedným procesom pri veľkosti matíc 1024x1024. Pri veľkostiach matíc 128x128 až 256x256 sú časi výpočtov vďaka vysokej réžií takmer identické.

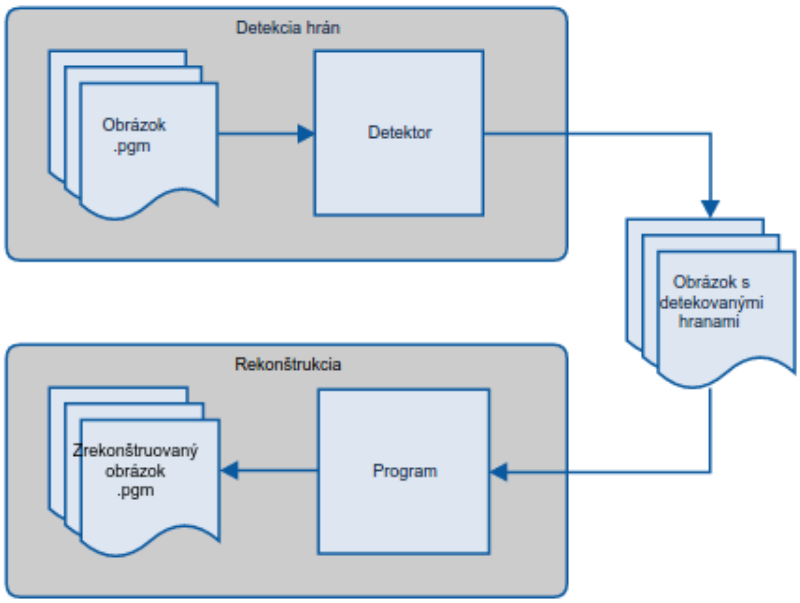


Obr. 5.9: Metóda rozdelenia matice A na riadky. Z grafu vyplýva, že rozdelenie matice A na riadky (1D - dekompozícia) je z pohľadu času efektívnejšie, ako rozdelenie na submatice obr. č. 5.8, pretože nieje potrebná komunikácia medzi procesmi, ako v prípade 2D - Dekompozície. S rastúcim počtom procesov, lineárne klesá čas potrebný na výpočet. Ďalej môžeme usúdiť, že zvyšovanie počtu procesov má zmysel pri väčších rozmeroch matíc (1024x1024 viď. graf).

Kapitola 6

Návrh riešenia rekonštrukcie obrazu

Vstupom programu pre rekonštrukciu obrazu bude obrázok, v ktorom sú detekované hrany, preto bude potrebné navrhnúť program, ktorý detekuje hrany zo vstupného obrázku. Diagram na obrázku č. 6.1 zobrazuje návrh rekonštrukcie obrazu.



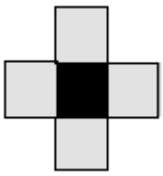
Obr. 6.1: Diagram návrhu rekonštrukcie obrazu. Návrh sa skladá z dvoch logických častí: Detekcia hrán a rekonštrukcia obrazu. Vstupom detekcie hrán je obrázok formátu PGM. Výstupom detekcie hrán a zároveň vstupom rekonštrukcie je obrázok formátu PGM s detekovanými hranami. Výstupom rekonštrukcie obrazu je pôvodný obrázok (zo vstupu detektora hrán).

6.1 Detektor hrán

Detektor hrán je program, ktorého vstupom je bežný obrázok a výstupom je obrázok s detekovanými hranami. Obidva obrázky sú vo formáte PGM (podkapitola 2.4). Detekcia hrán prebieha pomocou vzorca 6.1. Nová hodnota pixelu (*out*) je vypočítaná na základe počítania hodnôt zo štvor-okolia pixelu (*in*) vstupného obrázka detektoru. Za štvor-okolie daného pixelu sú považované pixely susediace s ním v horizontálnom a vertikálnom smere, viď obr. č. 6.2.

V rámci programu, ktorý rieši detekciu hrán bude implementovaná aj sekvenčná verzia rekonštrukcie obrazu založená na iteračnej metóde popísanej v podkapitole 6.2.

$$out_{i,j} = in_{i,j-1} + in_{i,j+1} + in_{i+1,j} + in_{i-1,j} + 4 \times in_{i,j} \tag{6.1}$$



Obr. 6.2: Štvor-okolie: Hodnota pixelu (čierny pixel) je vypočítaná na základe jeho okolia (biele pixely) podľa vzorca 6.1.

6.2 Metóda rekonštrukcie

Vstupom rekonštrukcie obrazu bude výstupný obrázok z detektora hrán, čiže obrázok s detekovanými hranami vo formáte PGM. Ako metóda rekonštrukcie bola zvolená inverzná metóda k detekcii hrán podľa vzorca 6.2, kde nová hodnota pixelu (*new*) je vypočítaná na základe starých (*old*) hodnôt štvor-okolia a hodnôt z obrázka s detekovanými hranami (*edgeOUT*). Hodnota pixelu je vypočítaná na základe štvor-okolia pixelu a odpovedajúceho pixelu výstupného obrázka z detektora hrán. Keďže nové hodnoty počítame zo štvor-okolia, nastáva problém pri hranách (okrajoch) obrázka, z dôvodu chýbajúcich pixelov z vonkajšej strany hrán. Z tohoto dôvodu bude pole obsahujúce obrázok pre potreby rekonštrukcie na hranách rozšírené. Metóda je iteračná, čo znamená, že vzorec na výpočet novej hodnoty pixelu počíta v každom ďalšom kroku s novými (aktuálnymi) hodnotami okolitých pixelov. Výstupom rekonštrukcie je zrekonštruovaný obrázok zapísaný do súboru vo formáte PGM. Táto metóda je veľmi podobná veľkým HPC (high-performance computing) úlohám, ktoré riešia parciálne diferenciálne rovnice iteračnými algoritmami [17].

$$new_{i,j} = 0.25 \times (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edgeOUT_{i,j}) \tag{6.2}$$

6.3 Paralelná verzia rekonštrukcie

Vstupný obrázok bude v prvom kroku spracovaný podľa PGM formátu. Obrázok sa následne na základe geometrickej dekompozície (podkapitola 2.5) rozdelí na nezávyslé segmenty podľa počtu procesov. Jednotlivé procesy si každú iteráciu (iteračná metóda 6.2) budú vymieňať `Halo zóny` popísané v podkapitole 2.5, ktorých výmena je dôležitá pre správné zrekonštruovanie hraníc medzi procesmi (príklad rekonštrukcie bez výmeny `Haló zón` obr. č 7.5). I/O[1] je možné riešiť buď pomocou kolektívnej komunikácie (sekvenčne) alebo paralelne pomocou modulu `mpi4py`. Zápis do súboru bude implementovaný obidvoma spôsobmi (sekvenčne aj paralelne).

[1] Vstup/výstup

Kapitola 7

Implementácia rekonštrukcie obrazu

7.1 Sekvenčná verzia

Vstupom sekvenčného programu pre rekonštrukciu sú nasledujúce parametre:

- Ľubovoľný obrázok formátu PGM, ktorý bude daný na vstup detektora hrán.
- Názov výstupného súboru pre obrázok s detekovanými hranami.
- Názov výstupného súboru pre zrekonštruovaný obrázok.
- Počet iterácií pri rekonštrukcii.

Vstupné parametre programu sú spracované pomocou modulu `argparse` [10]. Jadrom programu je funkcia `main()`, ktorá pozostáva zo štyroch logických celkov:

1. Spracovanie vstupného obrázka (obr. č. 7.1) - pomocou modulu `re` [10] je zo vstupného obrázka extrahovaná hlavička obsahujúca údaje o obrázku (body 1. - 7. zo štruktúry formátu PGM v podkapitole 2.4) a samotný obrázok uložený do dátovej štruktúry typu zoznam.

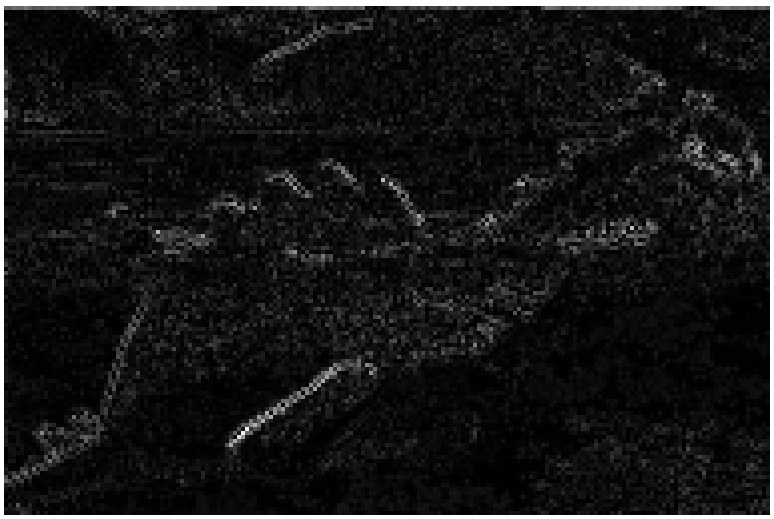
2. Detektor hrán - vstupom detektoru hrán je len samotný obrázok (bez hlavičky). Je vytvorený prázdny zoznam o veľkosti obrázka získaného z hlavičky, ktorý sa naplní vypočítanými hodnotami zo štvor-okolia vstupného obrázka podľa vzorca 6.1. Výstupný obrázok je uložený do súboru pre použitie v paralelnej verzii programu (podkapitola 7.2).

3. Rekonštrukcia - výstupný obrázok z detektora hrán (obr. č. 7.2) je vstupom pre rekonštrukciu. Pre výsledný zrekonštruovaný obrázok je vytvorená dátová štruktúra typu zoznam, avšak tento zoznam je väčší oproti rozmerom obrázka kvôli výpočtu okrajových hodnôt (pri výpočte hodnoty pixelu zo štvor-okolia musí byť pole väčšie aby sme nevyšli mimo obrázok). Kedže metóda rekonštrukcie je inverzná k metóde detekcie hrán, podľa počtu zadaných iterácií je opakovaný nasledovný krok (algoritmus č. 3) pre výpočet novej hodnoty pixelu vypočítaný podľa vzorca 6.2. Jednou z možností rekonštrukcie bolo využitie takzvaného medzníka (anglicky: *threshold*), kedy počet iterácií by nebol pevne daný, ale na základe zvoleného medzníku by sa každú iteráciu

počítal rozdiel medzi starou a novou hodnotou pixelu, až pokým by rozdiel hodnôt nedosiahol medzníka. Toto riešenie však nebolo príliš vhodné pre účel testovanie, kde je výhodnejšie poznať pevne daný počet iterácií, aj keď to nemusí byť z hľadiska presnej rekonštrukcie optimálne (príliš veľký počet zbytočných iterácií).

4. Zápis obrázka - nasleduje zápis zrekonštruovaného obrázka do zadaného súboru. K tomuto slúži funkcia `writeToFile()`, ktorej vstupom je zoznam obsahujúci jednotlivé pixely obrázka a jeho hlavička. Ako prvá je zapísaná hlavička obrázka a následne samotný obrázok podľa dokumentácie PGM formátu.

Obr. 7.1: Vstupný obrázok pre detektor hrán.



Obr. 7.2: Obrázok detekovaných hrán - výstup detektora hrán a vstup rekonštrukcie.

Algoritmus 3: Sekvenčná rekonštrukcia obrazu.

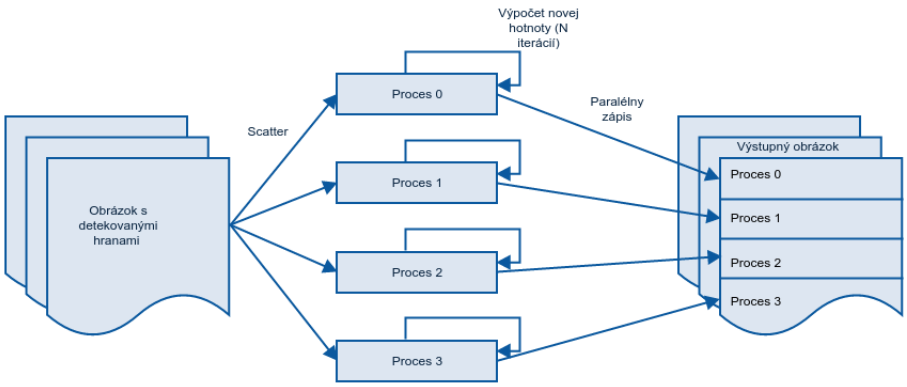
```
1 for x in iterations do
2     for j in width do
3         for i in height do
4             new[i,j] = 0.25 * (old[i-1,j] + old[i+1,j] + old[i,j-1] + old[i,j+1] -
                edgeOUT[i,j])
5         end
6     end
7 end
```

7.2 Paralelná verzia

Diagram paralelnej verzie rekonštrukcie je znázornený na obrázku č. 7.3. Paralelizácia rekonštrukcie obrazu využíva štandart MPI (modul `mpi4py`). Vstupné parametre (spracované modulom `argparse`) programu:

- Obrázok, obsahujúci detekované hrany.
- Názov výstupného súboru, kde bude uložený zrekonštruovaný obrázok.
- Počet iterácií pri rekonštrukcii.

Pre prácu s dátami bola na základe výsledkov mikrotestov (kapitola 5) zvolená štruktúra `numpy` polí.



Obr. 7.3: Diagram rekonštrukcie obrazu.

Rekonštrukcia

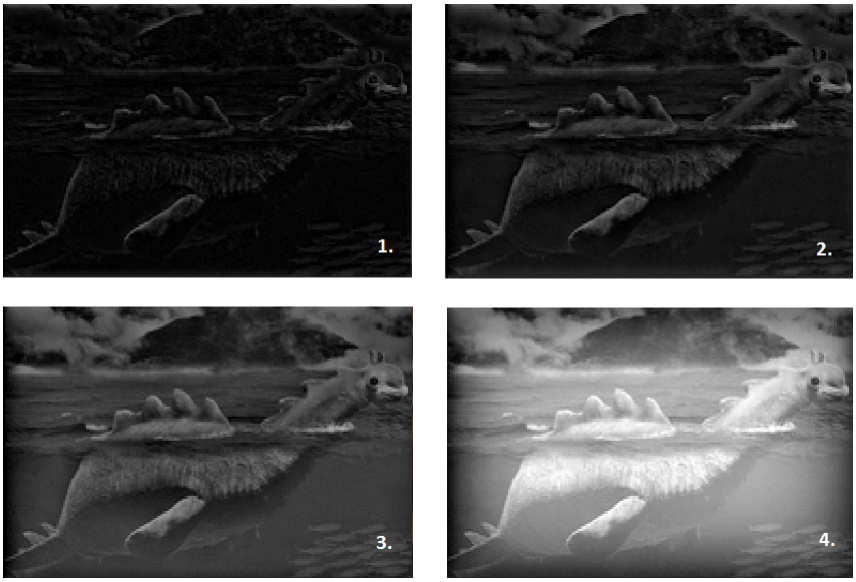
Postup pri paralelnej rekonštrukcii obrázka (obr. č. 7.3):

1. Spracovanie vstupného obrázka hlavným procesom (obdobne, ako pri sekvenčnej verzii).

2. Rozdelenie obrázka (funkcia `Scatter()`) hlavným procesom medzi všetky procesy po riadkoch (1D dekompozícia).

3. Výpočet nových hodnôt pixelov (obdobne, ako v sekvenčnej verzii).

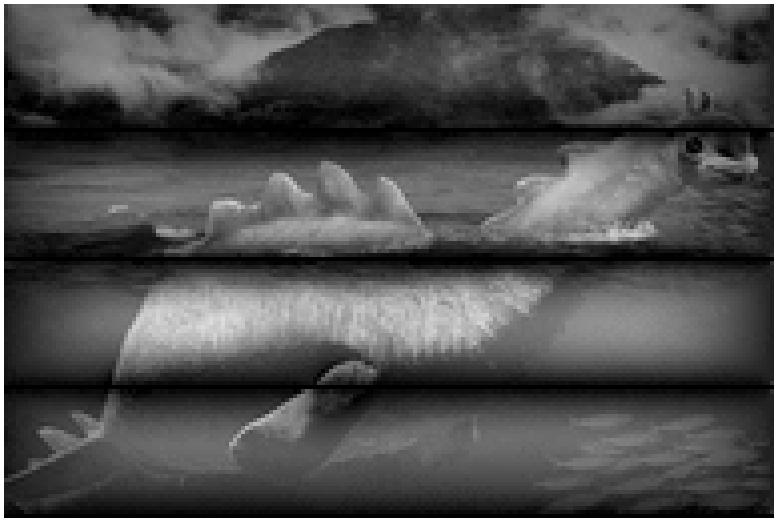
4. Po každej iterácii (priebeh rekonštruovania je zobrazený na obr. č. 7.4) riadiacej sa algoritmom č. 4 si procesy medzi sebou vymieňajú dáta pomocou `Haló zón`. Veľkosť použitého prekryvu (dáta, ktoré procesy medzi sebou zdieľajú výmenou `Haló zón`) má hodnotu jedného riadku matice reprezentujúcej obrázok. Komunikácia medzi procesmi je implementovaná pomocou neblokujúcej komunikácie (funkcie `Isend()/Irecv()`). V prípade neimplementovania komunikácie medzi procesmi, obrázok neni zrekonštruovaný správne a vyzerá ako na obrázku č. 7.5.



Obr. 7.4: Stav priebehu rekonštrukcie obrazu v rôznom počte iterácií. Stav rekonštrukcie bol zachytený pri nasledujúcich počtoch iterácií: 1. 5 iterácií, 2. 25 iterácií, 3. 60 ierácií, 4. 250 iterácií.

Algoritmus 4: Paralelná rekonštrukcia obrazu.

```
1 for x in iterations do
2     for j in width do
3         for i in height/numberOfProcess do
4             new[i,j] = 0.25 * (old[i-1,j] + old[i+1,j] + old[i,j-1] + old[i,j+1] -
                edgeOUT[i,j]);
5             posielanie Haló zón medzi procesmi pomocou MPI komunikácie;
6         end
7     end
8 end
```



Obr. 7.5: Zrekonštruovaný obrázok bez komunikácie medzi procesmi. Čierne čiary reprezentujú oblasti medzi procesmi, kde na základe chýbajúcej komunikácie, chýbajú aktuálne dáta pre správnu rekonštrukciu.

Zápis do súboru

Zápis obrázka do súboru prebiehal niekoľkokrát počas rekonštrukcie kvôli kontrolným výpisom. Z tohoto dôvodu bol implementovaný nielen sekvenčne, ale aj paralelne (na rozdiel od načítania súboru - načítanie prebehlo len jedenkrát hlavným procesom):

- Paralelný zápis do súboru [4]. Pomocou funkcie `Gather()` hlavný proces zozbiera od ostaných údaje o veľkosti svojich lokálnych zrekonštruovaných obrázkov a vypočíta celkovú veľkosť obrázka potrebnú pre paralelný zápis. Definuje sa pomocou metódy `Create_subarray` typ poľa `MPI.CHAR`. Tento typ definuje celkovú veľkosť poľa (zrekonštruovaného obrázka), veľkosť čiastočných (lokálnych) polí jednotlivých procesov a index od ktorého daný proces má do súboru zapisovať. Na základe tohoto vytvoreného typu je nastavený pohľad pomocou metódy `MPI.File.Set_view`. Následne je spustený paralelný zápis metódou `MPI.File.Write_all`.
- Sekvenčný zápis do súboru. Pre porovnanie bol implementovaný aj sekvenčný zápis rovnakých dát. Hlavný proces zozbiera lokálne dáta (funkcia `Gather()`) od všetkých procesov a obdobným spôsobom, ako v sekvenčnej verzii programu sú dáta pomocou funkcie `writeToFile()` zapísané do súboru.

7.3 Zložitosť algoritmov rekonštrukcie obrazu

Uvažujme sekvenčný algoritmus (algoritmus č. 3) rekonštrukcie obrazu, ktorého vstupom je pre jednoduchosť štvorcová matica o veľkosti $N \times N$, kde N je rozmer vstupu v jednej dimenzii. Jedna iterácia sekvenčného algoritmu prechádza postupne všetky prvky vstupnej matice ($N \times N$ prvkov) a počíta pre ne novú hodnotu. Výpočet hodnoty prebehne v konštantnom čase, celková časová zložisť algoritmu je teda $t(N) = N \times N = \mathrm{O}(N^2)$.

Paralelný algoritmus (algoritmus č. 4) začína rozdelením vstupnej matice medzi P procesorov. Každý procesor p_i, kde i je 1-P procesorov, dostane N/P riadkov matice, celkovo

teda obrží $N \times (N/P)$ prvkov vstupnej matice. Samotný výpočet už prebieha rovnako ako sekvenčný algoritmus s tým rozdielom, že paralélne pracuje P procesorov (každý nad svojou submaticou). V jednej iterácii algoritmu teda každý procesor vypočíta $N \times (N/P)$ nových hodnôt výstupnej matice a pošle riadky na okrajoch susedným procesorom. Uvažujme, že komunikácia medzi procesormi prebehne v konštantnom čase. Časová zložitosť paralélneho algoritmu je teda $t(N) = N \times (N/P) = \mathrm{O}(N^2/P)$.

7.4 Správa kódu

Pri programovaní boli využité nasledujúce prostriedky:

- Zdrojové súbory boli zdokumentované pomocou `Doxygen` nástroja [1].
- Pre vytváranie verzii bol použitý nástroj `Git` [2]. Všetky zdrojové texty boli priebežne ukladané na projektový `GitLab` server (viac ako 60 "commitov").
- Od začiatku vývoja boli vedené na `GitLabovom` servery takzvané *Issues*, ktoré zachytávajú návrh, riešené problémy a výsledky jednotlivých podúloh vyvíjaných v rámci tejto bakalárkej práci.

Kapitola 8

Testovanie

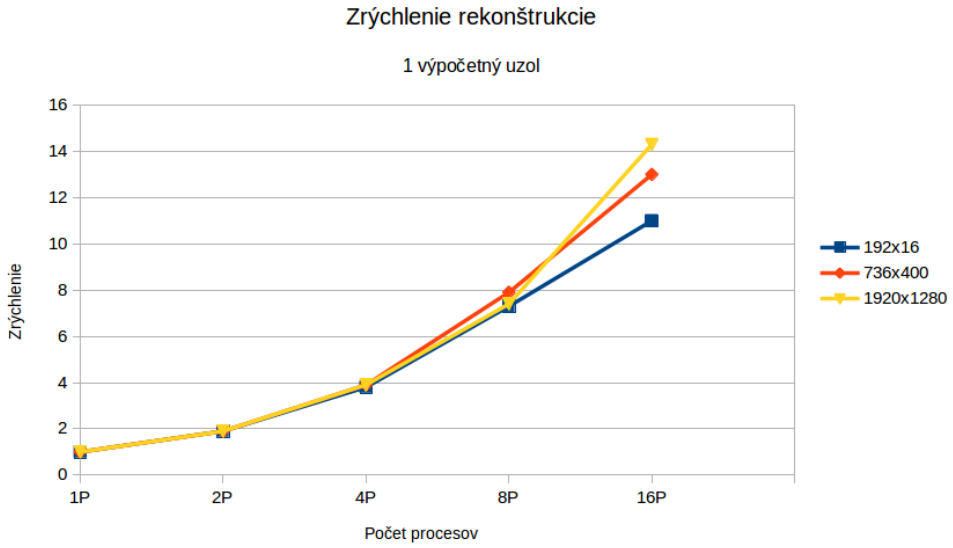
Merania prebiehali na superpočítači Anselm. Príprava prostredia:

- Prihlásenie sa na login uzol superpočítača Anselm.
- Spustenie interaktívneho terminálu s požadovanými prostriedkami (počet procesov, fronta, čas). Príklad spustenia:
  qsub -A OPEN-7-15 -q qexp -l select=2:ncpus=16:mpiprocs=1 -I
- Načítanie modulu umožňujúceho prácu s rozhraním MPI a jazykom Python:
  `ml ScientificPython`.
- Zahájenie meraní.

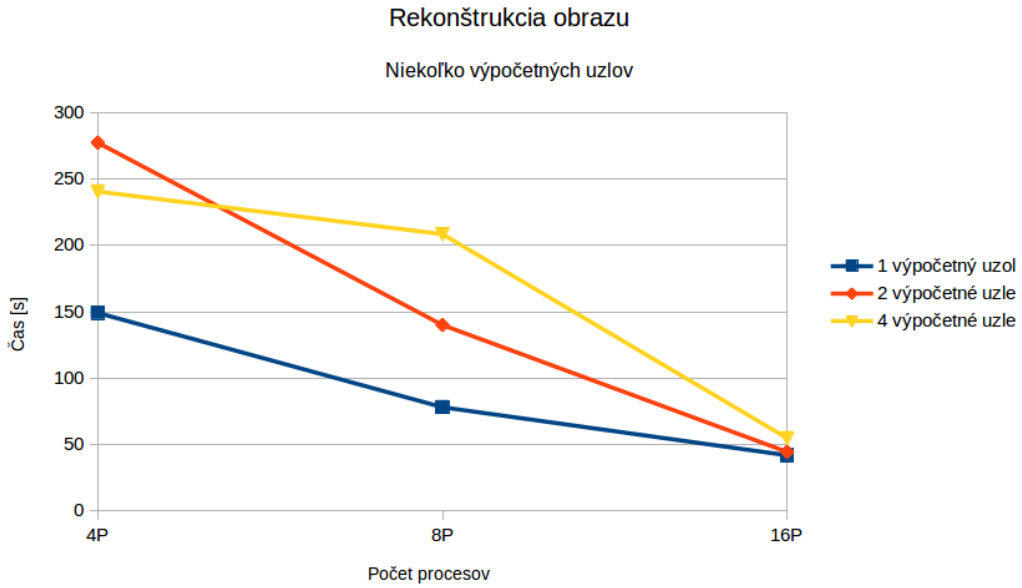
8.1 Meranie zrýchlenia rekonštrukcie

Meranie zrýchlenia rekonštrukcie bolo testované pri použití rôzneho počtu procesov (1, 2, 4, 8, 16) a veľkostí obrázkov (192x16 - 1920x5120). Merania prebiehali v rámci 1 až 4 výpočetných uzlov. Pri testovaní bol kladený dôraz na otestovanie rozdielov medzi malými a veľkými obrázkami z hľadiska efektivity množstva použitých procesov, čiže práce na jeden proces vzhľadom na potrebnú réžiu (komunikáciu) medzi procesmi. Medzi očakávané výsledky meraní patril predpoklad, že pri malých obrázkoch bude pri väčšom množstve použitých procesov príliš vysoká réžia na úkor efektivity.

Z meraní, ako je možné vidieť na obr. č. 8.1 vyplýva, že zrýchlenie sa pri rekonštrukcii v rámci jedného výpočetného uzla zvyšuje rýchlejšie pre väčšie obrázky, pretože procesy majú viac práce vzhľadom na potrebnú réžiu medzi nimi. Pri použití viacerých výpočetných uzlov sa čas rekonštrukcie spomaľuje, pretože komunikácia medzi procesmi bežiacimi na rozdielnych výpočetných uzloch je pomalšia ako v rámci jedného výpočetného uzla. Viacej výpočetných uzlov bolo zmeraných na rozmeroch obrázku 1920x640 pri počte uzlov 2 a 4, čo možno vidieť na grafe č. 8.2.



Obr. 8.1: Graf závislosti zrýchlenia rekonštrukcie troch veľkostí obrázkov vzhľadom na počet použitých procesov v rámci jedného výpočetného uzla. Zrýchlenie pri menšom obrázku je menšie, ako pri veľkých obrázkoch z dôvodu menšieho množstva práce na proces vzhľadom na réžiu medzi procesmi.



Obr. 8.2: Graf porovnania rýchlosti rekonštrukcie obrazu (veľkosť 1920x640) pri použití viacerých výpočetných uzlov. Z grafu vyplýva, že so zvyšujúcim sa počtom komunikujúcich uzlov narastá čas potrebný na rekonštrukciu obrazu.

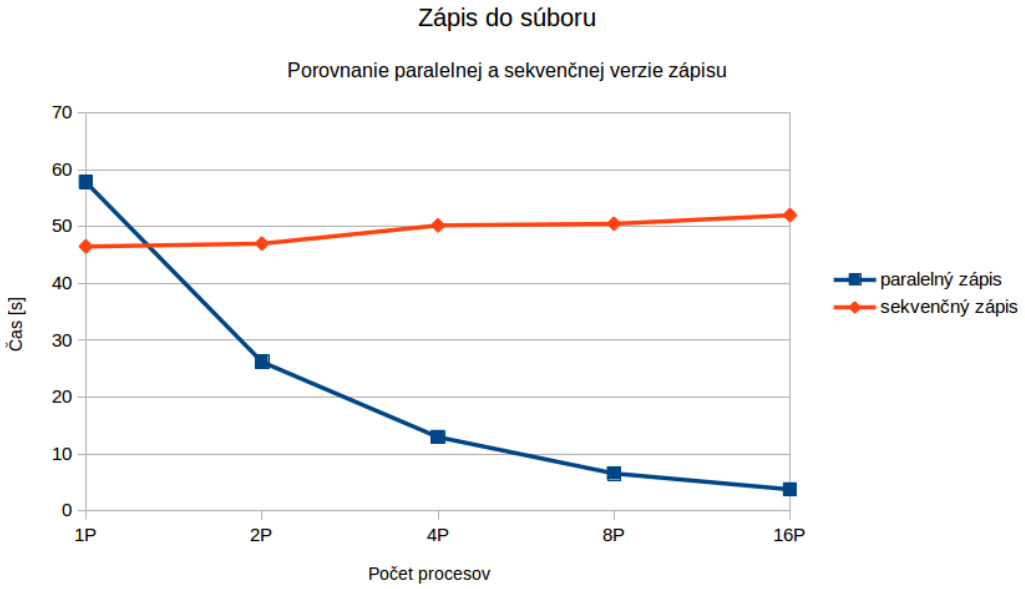
8.2 Meranie zápisu do súboru

Pri testovaní zápisu do súboru boli porovnávané dve techniky:

- Paralelný zápis: využitie funkcií knižnice `mpi4py`.
- Sekvenčný zápis: mnou naimplementovaný sekvenčný algoritmus, kde zapisuje len jeden proces.

Meraný bol čas potrebný pre zápis obrazu pri rôznych počtoch použitých procesov pri veľkosti obrazu 1920x5120 bodov. Očakávaným výsledkom bol predpoklad, že paralelný zápis bude rýchlejší priamo úmerne počtu zapisujúcich procesov.

Z výsledkov meraní, ktoré sú zaznamenané na obr. č. 8.3 je možné usúdiť, že pri paralelnom zápise priamo úmerne klesá čas potrebný pre zápis obrázka vzhľadom na počet použitých procesov. Pri sekvenčnom zápise dát naopak čas zápisu mierne stúpa so zvyšujúcim počtom použitých procesov z dôvodu zvýšenia réžie - potreby zozbierania dát od všetkých procesov (funkcia `Gather()`). Pri zápise pomocou jedného procesu je sekvenčná verzia 1,2 krát rýchlejšia ako paralelná, pretože pri paralelnej verzii sa vytvára nový podtyp, čo je pri zápise jedným procesom zbytočná operácia.



Obr. 8.3: Graf porovnania rýchlosti zápisu dát pomocou sekvenčnej a paralelnej verzie zápisu, pri použití 1-16 procesov. V rámci jedného výpočetného uzla je výhodnejší paralelný zápis, kedže čas potrebný na zápis priamo úmerne klesá so zvyšujúcim sa počtom použitých procesov, zatiaľ čo pri sekvenčnom zápise naopak potrebný čas na zápis stúpa. Sekvenčný zápis je výhodný len pri použití jedného procesu, kedy paralelný zápis stráca význam.

Kapitola 9

Záver

Hlavnými cieľmi práce boli implementácia sady mikrotestov na získanie základných poznatkov paralelného spracovávania, neskôr rekonštrukcia obrazu. Ďalším z cieľov bolo porovnať paralelné programy v programovacích jazykoch Python a C z hľadiska efektivity a náročnosti implementácie.

Na základe vyhodnotetia mikrotestov bola zvolená na prácu s dátami štruktúra `NumPy` polí, keďže v testoch obstála s najrýchlejšími časmi preposielania dát. Ďalej z výsledkov vyplynulo, že program v jazyku C beží v niektorých prípadoch až 2-krát rýchlejšie, avšak v jazyku Python sa rýchlejšie vytvárajú prototypy a je jednoduchší na pochopenie. Preto aj rekonštrukcia obrazu bola implementovaná v jazyku Python. Všetky získané poznatky z mikrotestov boli aplikované na probléme paralelnej rekonštrukcii obrazu.

V rámci hlavnej úlohy bolo cieľom paralelne zrekonštruovať zadaný obrázok do pôvodného stavu. Navyše bolo potrebné navrhnúť a implementovať program na detekciu hrán, ktorého výstup bol vstupom rekonštrukcie. Keďže rekonštrukcia prebiehala v niekoľkých iteráciách, bola výpočetne veľmi náročná. K dodaniu dostatočného výkonu poslúžil superpočítač Anselm. Detekcia hrán a rekonštrukcia bola implementovaná v sekvenčnej aj paralelnej verzii. Ďalšou časovo náročnou operáciou, ktorá bola implementovaná paralelne, bol zápis do súboru.

Zo záverečných testov vyplynulo, že pri zvyšovaní počtu procesov dochádza priamo úmerne k zrýchleniu rekonštrukcie a zápisu obrázka, ale iba do momentu kedy práca na proces je príliš malá vzhľadom na réžiu potrebnú pre komunikáciu s ostatnými procesmi. Testy potvrdili zrýchlenie paralelnej rekonštrukcie obrazu voči sekvenčnej. Práca potvrdila možné uplatnenie jazyka Python pre ľudí, ktorý nemajú dostatočné skúsenosti s programovaním výpočtovo náročných aplikácií (napr. vedci). Naviac jazyk Python poskytuje veľké množstvo knižníc, obsahujúcich hotové riešenia, ktoré prispievajú k rýchlemu vytváraniu prototypov. Samozrejme jazyk Python je vhodný len pre úlohy, u ktorých nieje požiadavkou maximálne využitie dostupných prostriedkov (v týchto prípadoch je potrebné použiť jazyk nižšej úrovne, napr. C).

Veľký prínos malo vytvorenie tejto práce aj pre mňa osobne, keďže v dnešnej rýchlej dobe je riešenie paralelizácie problémov aktuálnou témou. V práci by som rád pokračoval v rámci vylepšenia aktuálneho riešenia, neskôr implementáciou komplexnejších úloh využívajúcich štandartu MPI.

Literatúra

[1] *Doxygen*. [Online].[vid. 4.4.2017].
URL https://www.stack.nl/~dimitri/doxygen/manual/docblocks.html

[2] *Git*. [Online].[vid. 4.4.2017].
URL https://git-scm.com/documentation

[3] *Hardware Overview*. [Online].[vid. 2.1.2017].
URL https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview

[4] *Modul mpi4py*. [Online].[vid. 4.4.2017].
URL http://mpi4py.scipy.org/docs/usrman/

[5] *MPI: A Message-Passing Interface Standard*. [Online].[vid. 30.12.2016].
URL http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[6] *MPI_Barrier*. [Online].[vid. 30.12.2016].
URL https://www.open-mpi.org/doc/v1.5/man3/MPI_Barrier.3.php

[7] *MPI_Reduce*. [Online].[vid. 30.12.2016].
URL http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

[8] *NumPy*. [Online].[vid. 21.1.2017].
URL http://www.numpy.org/

[9] *PGM Formát obrázku*. [Online].[vid. 25.4.2017].
URL http://netpbm.sourceforge.net/doc/pgm.html

[10] *Python 2.7.13 documentation*. [Online].[vid. 30.12.2016].
URL https://docs.python.org/2/

[11] *What is Python? Executive Summary*. [Online].[vid. 29.12.2016].
URL https://www.python.org/doc/essays/blurb/

[12] Barlas, G.: *Multicore and GPU Programming*. Morgan Kaufmann, december 2014, ISBN 9780124171374.

[13] Dvořák, V.: Architektury a programování paralelních systémů. [Prednáška01].[vid. 30.12.2016].

[14] Dvořák, V.: Architektury a programování paralelních systémů. [Prednáška02].[vid. 31.12.2016].

[15] Gupta, A.; Kumar, V.: *Scalability of Parallel Algorithms for Matrix Multiplication*. IEEE, 1993, ISBN 0849389836.

[16] Rouse, M.: supercomputer. [Online].[vid. 2.1.2017].
URL http://whatis.techtarget.com/definition/supercomputer

[17] Saad, Y.: *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, druhé vydání, april 2003, ISBN 9780898715347.

Príloha A

Obsah CD

V nasledujúcej prílohe je popísaná adresárová štruktúra spolu s obsahom priloženého CD.

- **mikrotesty/** - adresár obsahujúci zdrojové súbory mikrotestov
- **rekonstrukcia/** - adresár obsahujúci zdrojové súbory rekonštrukcie obrazu
- **dokumentacia/** - adresár s vygenerovanou dokumentáciou pomocou doxygen
- **technicka sprava/** - adresár obsahujúci zdrojové súbory k tejto technickej správe (bakalárskej práci)
- **Doxyfile** - konfiguračný súbor pre doxygen
- **README** - súbor obsahujúci návod, ako pracovať so zdrojovými kódmi